个性化你的图床

编程狂人

Programming Madman

NO.23

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法,从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道,它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:

http://www.tuicool.com/mags/5365fad9d91b145e3d008163

欢迎下载推酷客户端体验更多阅读乐趣

Lucene 4.8.0 发布了，变化一如既往的大，新特性一一解读

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do tempor incididunt ut labore et dolore magna aliqua.

作者:accesine960

10年之前，你是1.0； 10年之后，你是4.8。放在10年这个时间跨度上看，也许变化就没那么大了。

看看这次发布有哪些变化吧:

1、Apache Lucene 现在要求Java的最低版本为： Java 7 , update 55 ； 推荐使用 Oracle Java 7 或 OpenJDK 7 ， 之前版本的JVM bug 会影响到lucene。

2、Apache Lucene全面兼容 Java 8。

3、所有的索引文件开始存储checksums，在索引合并和读取的时候进行有效性检查。减少出现某个索引文件（物理）损坏带来的问题，主要是针对硬件或者JVM bug 引起的索引损坏。

4、提供了针对第一次搜索结果集合的重打分（权重调整）API；相当于对搜索结果的二次自定义排序。

5、AnalyzingInfixSuggester 类提供了支持NRT的自动建议功能。

6、把基于批量处理的打分过程 bulk scoring 和基于迭代的打分过程分离了，这对于批量打分的过程更高效一些。

7、在建立索引的时候针对Hash term 使用了 MurmurHash3 的hash方法，很高效的方法 。

http://zh.wikipedia.org/wiki/Murmur%E5%93%88%E5%B8%8C

8、 IndexWriter现在支持更新二进制类型的字段了

9、优化了 HunspellStemFilter 占用内存的大小（10至100倍的减少）

Hunspell 是一种检查拼写spellcheck流行的方法， OpenOffice中就用了它来进行拼写检查。

HunspellStemFilter 是TokenFilter的扩展，可以用这个算法来过滤词的不同变形（时态，语气等）。

中文的Token应该享受不到这个特性。

http://en.wikipedia.org/wiki/Hunspell

10、Lucene现在使用Java 7中的文件系统函数，比如即使在索引打开的时候，也可以删除索引文件。

11、修复了NativeFSLockFactory 中的一个严重的bug： 允许多个IndexWriter获得一个lock。

所以强烈建议升级到 lucene 4.8 。

原文链接:http://blog.csdn.net/accesine960/article/details/24741703

MongoDB和Cloudera结盟，欲征服大数据市场

翻译/毛梦琪 责编/魏伟

摘要：MongoDB是NoSQL市场上成功的数据库供应商，而Cloudera在Hadoop市场上也是领袖级的大公司，近日，两家公司提出要共享营销和销售渠道，声称目的只有一个：为客户提供大数据整体解决方案，消除客户的疑虑。

【编者按】近日，MongoDB和Cloudera宣布结盟，两家公司致力于整合NoSQL和Hadoop技术，促使两家公司形成优势互补的良好合作关系，它们描绘了一幅“NoSQL应用于操作型数据库，Hadoop应用于数据分析”的大数据市场规划图，给NoSQL和Hadoop提供了一个清晰的定位，此外大额风险资本的注入也加强了这两家公司征服大数据市场的信心，下面看作者Doug Henschen为我们带来的精彩报道。

以下为译文：

MongoDB和Cloudera，分别是NoSQL市场和Hadoop市场的重量级大公司。近日，两家公司提出要共享营销和销售渠道，声称目的只有一个：为客户提供大数据整体解决方案，消除客户的疑虑。

MongoDB是NoSQL市场上成功的数据库供应商，而Cloudera在Hadoop市场上也是领袖级的大公司，两家公司都认识到，目前客户对大数据还很困惑，如果能为客户解除这些疑虑，为客户提供整体的解决方案，对两家公司未来的发展都是极为有利的。

这两家公司在周二宣布了结盟的消息，它们致力于建立更深的合作伙伴关系，作为合作伙伴关系的一部分，MongoDB和Cloudera将把它们的产品整合营销和出售，在大数据技术上形成互补效应，简单地说，MongoDB将被定位成面向高扩展性应用的操作型数据库，而Cloudera基于Hadoop的企业数据中心将被用作分析平台。

Matt，MongoDB副总裁，负责MongoDB的市场、业务发展和企业战略。在一次电话采访中，他告诉我们：“在去年Strata会议上，我讲过MongoDB数据库和Hadoop平台应该结合在一起，那时我就认识到MongoDB需要在战略上做出一些改变。当时，好多人不理解，他们认为MongoDB和Hadoop是竞争对手的关系。”

Yuri Bukhan，Cloudera的ISV联盟项目负责人，他告诉我们：“你或许会觉得区分NoSQL和Hadoop并不难，那些不知道怎样合理使用NoSQL和Hadoop的人应该去做更多的研究，但事实上，在这两个平台之间确实有许多灰色地带，比如：HBase，其实HBase就是Hadoop中的NoSQL数据库，但是HBase更适合于超大规模却相对简单的用例，而MongoDB支持更加复杂的数据建模。”

Bukhan引用了在线行为分析，以比较HBase和MongoDB所担当的不同角色，以及发挥的不同作用。“比如，当你研究简单的用户点击或者会话的时候，HBase可以提供非常快速的随机读取和写入，你可以基于特定的关键值对用户进行查找等操作，而MongoDB可以为你提供更丰富的模型，使你可以通过线上应用全程追踪用户的行为。”

目前，MongoDB和Cloudera已经有了双向的数据连接，但从Asay和Bukhan那里了解到，两家公司还在准备将MongoDB和Hadoop更好地整合到一起，借此实时操作性数据利用MongoDB可以在Couldera数据中心中建立快照，用于并行分析。这样的分析接近实时，通过Shark框架或者Impala传递回MongoDB，接着触发个性化内容的展示或者产生一个基于Hadoop分析的最合适产品。

集成之后的产品，据估计会在六月纽约的MongoDB World中展出，该产品将运行在YARN上，新的资源管理层中引入了Hadoop 2.0。过去很难想象MongoDB和Cloudera会运行在同一个服务器集群上，当时很多人担心MongoDB和Cloudera和产生冲突。

如今，MongoDB和Cloudera建立了合作伙伴关系，很多问题都会得以解决，两个成功的公司将描绘出一幅“NoSQL应用于操作型数据库，Hadoop应用于分析”这样的大数据市场规划图。有人一定会问：为什么选择Cloudera而不是整个Hadoop社区？

Asay指出：“这就是开源的好处之一，开源社区内，技术在不断更新和发展，我们的很多技术都有很强的适用性，而且是公开的，所以其他的Hadoop供应商也能够使用这些技术。”

其他的NoSQL供应商，像DataStax这样的公司，没能在NoSQL和Hadoop之间划出一条清晰的界限。比如DataStax的软件发行版，其中既包括Cassandra的NoSQL数据库又包括了Hadoop，它们共同运行在同一个集群上，而且，DataStax和其他高扩展性数据库供应商一直忙于加强和兜售它们数据库的分析和查询性能。

据Asay所说，MongoDB和Cloudera联合销售软件的同时，它们将各自的销售能力结合到一起为它们的产品提供最好的、持续的支持。一段时间以后，大数据市场的局势可能会变得比现在更加复杂，但是由于已经拥有了大额风险资本的注入，MongoDB和Cloudera对未来征服大数据市场很有信心。

原文链接：MongoDB, Cloudera Form Big Data Partnership

原译文链接:http://www.csdn.net/article/2014-04-30/2819570-BigData-MongoDB-Cloudera

JavaScript 单元测试框架：Jasmine 初探

作者:常文昭,软件工程师,IBM

简介

随着互联网浪潮的逐渐兴起，各种基于互联网的云战略也不断涌现，各个公司对云平台的理解和实现不尽相同，而云+端的模式越来越多受到关注。其中的端可以理解为终端用户手中的各种终端，包括 PC、手机、平板等不一而足。 而越来越多的用户愿意在自己的设备上使用轻量级的基于浏览器的应用。这类应用的安装部署可以是通过插件的方式安装，也有可能是直接以网页的形式访问而无需安装，相对于富客户端的下载安装，对用户来说更加简单方便，用户体验也更好。

这类应用对开发人员来说，需要一些互联网相关的技术，其中必不可少 HTML CSS 和 JavaScript 技术。而 JavaScript 作为一种客户端脚本语言，和传统编程语言 Cpp、Java 等相比，没有诸如 Eclipse、Visual Studio 等集成开发调试环境，其调试和测试是对开发人员都是一项挑战。

目前 JS 单元测试框架有丰富的选择，比如 Buster.js、TestSwarm、JsTestDriver 等。而 Jasmine 作为流行的 JavaScript 测试工具，很轻巧只有 20K 左右，而功能丰富，让我们可以容易的写出清晰简洁的针对项目的测试用例。对基于 JavaScript 开发的项目来说，是一款不错的测试框架选择。

搭建环境

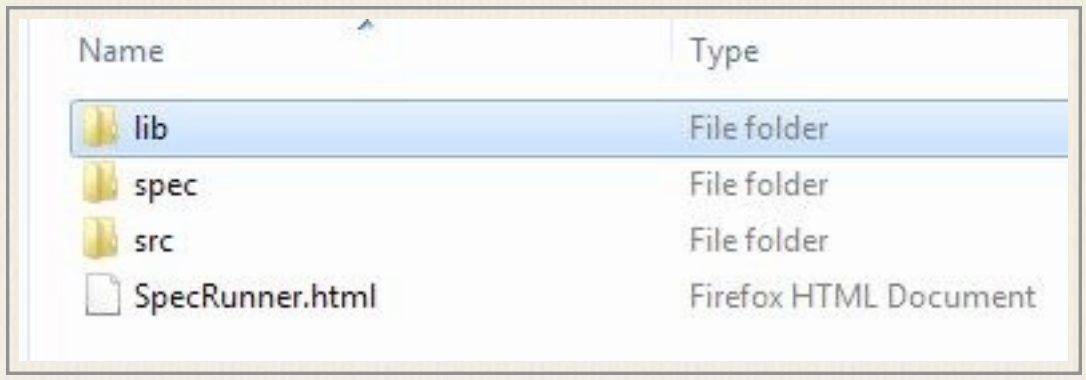
获取安装包

可以在开源社区网站下载最新的 Jasmine 安装包, 目前的 Standalone 的最新版本是 1.3.0. 下载地址https://github.com/pivotal/jasmine/downloads

配置安装:

下载后的.zip 文件包解压缩，如下的目录结构:

图 0.目录结构



其中 lib 文件夹中包含 Jasmine 的源代码。采用如下相对路径可以包含 Jasmine，进而开发基于 Jasmine 的测试用例。

```
<link rel="shortcut icon" type="image/png"
href="lib/jasmine-1.3.0/jasmine_favicon.png">
<link rel="stylesheet" type="text/css"
href="lib/jasmine-1.3.0/jasmine.css">
<script type="text/javascript"
src="lib/jasmine-1.3.0/jasmine.js"></script>
<script type="text/javascript"
src="lib/jasmine-1.3.0/jasmine-html.js"></script>
```

spec 和 src 和 SpecRunner.html 是 Jasmine 的一个完整示例，用浏览器打开 SpecRunner.html，即可看到执行的结果。

基本概念

describe

describe 是 Jasmine 的全局函数，作为一个 Test Suite 的开始，它通常有 2 个参数：字符串和方法。字符串作为特定 Suite 的名字和标题。方法是包含实现 Suite 的代码。

清单 1.测试用例

```
describe("This is an exmaple suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
    expect(false).toBe(false);
    expect(false).not.toBe(true);
  });
});
```

Specs

Specs 通过调用 it 的全局函数来定义。和 describe 类似，it 也是有 2 个参数，字符串和方法。每个 Spec 包含一个或多个 expectations 来测试需要测试代码。

Jasmine 中的每个 expectation 是一个断言，可以是 true 或者 false。当每个 Spec 中的所有 expectations 都是 true，则通过测试。有任何一个 expectation 是 false，则未通过测试。而方法的内容就是测试主体。

JavaScript 的作用域的规则适用，所以在 describe 定义的变量对 Suite 中的任何 it 代码块都是可见的。

清单 2.测试用例

```
describe("Test suite is a function.", function() {
  var gVar;
  it("Spec is a function.", function() {
    gVar = true;
    expect(gVar).toBe(true);
  });
 it("Another spec is a function.", function() {
```

```
    gVar = false;
    expect(gVar).toBe(false);
  });
});
```

Expectations

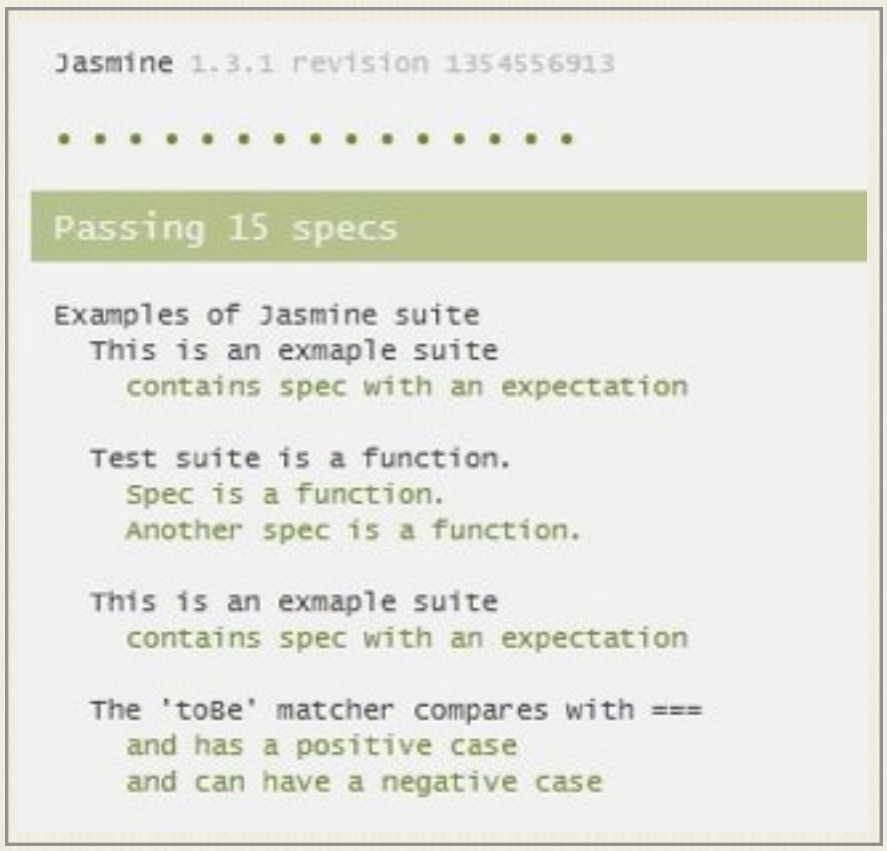
Expectations 是由方法 expect 来定义，一个值代表实际值。另外的匹配的方法，代表期望值。

清单 3.测试用例

```
describe("This is an exmaple suite", function() {
  it("contains spec with an expectation", function() {
    var num = 10;
    expect(num).toEqual(10);
  });
});
```

以上代码可以在附件中的 List.html 运行，结果见下图：

图 1.测试用例结果



总结

describe 方法用来组织相关的 Spec 集合。string 参数作为 Spec 集合的名字，会和其中的 Spec 连接组成 Spec 的完整名字。这样在一个大的 suite 中可以更容易找到某个 Spec。如果给它们命名适当，Specs 读起来是一个典型的 BDD 样式的句子。

Spec 是作为测试主体，Suite 是一个或多个 Spec 的集合。

describe 和 it 代码块中都是方法，可以包含任何可执行的代码来实现测试。而方法的内容就是 Suites。

常见用法

Matchers

每个 Matchers 实现一个布尔值，在实际值和期望值之间比较。它负责通知 Jasmine，此 expectation 是真或者假。然后 Jasmine 会认为相应的 spec 是通过还是失败。

任何 Matcher 可以在调用此 Matcher 之前用 not 的 expect 调用，计算负值的判断。

清单 4.测试用例

```
describe("The 'toBe' matcher compares with ===", function() {
  it("and has a positive case ", function() {
    expect(true).toBe(true);
  });
  it("and can have a negative case", function() {
    expect(false).not.toBe(true);
  });
});
```

Included Matchers

Jasmine 有很多的 Matchers 集合。下面的例子中举例说明一些常用的 Matchers。另外当项目需要特定的判断，而没有包含在 Jasmine 的 Matchers 时，也可以通过写定制的 Matchers 来实现.

清单 5.测试用例

```
describe("Included matchers:", function() {
  it("The 'toBe' Matcher", function() {
    var a = 3.6;
    var b = a;
    expect(a).toBe(b);
    expect(a).not.toBe(null);
  });
  describe("The 'toEqual' matcher", function() {
    it("works for simple literals and variables", function() {
  var a = "varA";
  expect(a).toEqual("varA");
});
it("Work for objects", function() {
  var obj = {
    a: 1,
    b: 4
  };
  var obj2 = {
    a: 1,
    b: 4
  };
```

```
      expect(obj).toEqual(obj2);
    });
  });
  it("The 'toBeDefined' matcher ", function() {
    var obj = {
      defined: 'defined'
    };
    expect(obj.defined).toBeDefined();
    expect(obj.undefined).not.toBeDefined();
  });
});
```

其他的 Matchers 还有：

toBe()

toNotBe()

toBeDefined()

toBeUndefined()

toBeNull()

toBeTruthy()

toBeFalsy()

toBeLessThan()

toBeGreaterThan()

toEqual()

toNotEqual()

toContain()

toBeCloseTo()

toHaveBeenCalled()

toHaveBeenCalledWith()

toMatch()

toNotMatch()

toThrow()

Setup and Teardown

为了使某个测试用例干净的重复 setup 和 teardown 代码， Jasmine 提供了全局的 beforeEach 和 afterEach 方法。正像其名字一样，beforeEach 方法在 describe 中的

每个 Spec 执行之前运行，afterEach 在每个 Spec 调用后运行。

这里的在同一 Spec 集合中的例子有些不同。测试中的变量被定义为全局的 describe 代码块中，用来初始化的代码被挪到 beforeEach 方法中。afterEach 方法在继续前重置这些变量。

清单 6.测试用例

```
describe("An example of setup and teardown)", function() {
  var gVar;
  beforeEach(function() {
    gVar = 3.6;
    gVar += 1;
  });
  afterEach(function() {
    gVar = 0;
  });
  it("after setup, gVar has new value.", function() {
```

```
    expect(gVar).toEqual(4.6);
  });
  it("A spec contains 2 expectations.", function() {
    gVar = 0;
    expect(gVar).toEqual(0);
    expect(true).toEqual(true);
  });
});
```

嵌套代码块

describe 可以嵌套， Specs 可以定义在任何一层。这样就可以让一个 suite 由一组树状的方法组成。在每个 spec 执行前，Jasmine 遍历树结构，按顺序执行每个 beforeEach 方法。Spec 执行后，Jasmine 同样执行相应的 afterEach。

清单 7.测试用例

```
describe("A spec", function() {
  var gVar;
  beforeEach(function() {
    gVar = 3.6;
    gVar += 1;
  });
  afterEach(function() {
    gVar = 0;
  });

  it("after setup, gVar has new value.", function() {
```

```
  expect(gVar).toEqual(4.6);
});
it("A spec contains 2 expectations.", function() {
  gVar = 0;
  expect(gVar).toEqual(0);
  expect(true).toEqual(true);
});
describe("nested describe", function() {
  var tempVar;
  beforeEach(function() {
    tempVar = 4.6;
  });
    it("gVar is global scope, tempVar is this describe scope.", function() {
      expect(gVar).toEqual(tempVar);
    });
  });
});
```

跳过测试代码块

Suites 和 Specs 分别可以用 xdescribe 和 xit 方法来禁用。运行时，这些 Suites 和 Specs 会被跳过，也不会在结果中出现。这可以方便的在项目中可以根据需要来禁用隐藏某些测试用例。

清单 8.测试用例

```
xdescribe("An example of xdescribe.", function() {
  var gVar;
```

```
  beforeEach(function() {
   gVar = 3.6;
   gVar += 1;
  });
  xit(" and xit", function() {
   expect(gVar).toEqual(4.6);
  });
});
```

以上代码可以在附件中的 List.html 运行，结果见下图：

图 2.测试用例结果

```
Included matchers:
  The 'toBe' Matcher

  The 'toEqual' matcher
    works for simple literals and variables
    Work for objects
  The 'toBeDefined' matcher

An example of setup and teardown
  after setup, gVar has new value.
  A spec contains 2 expectations.

A spec
  after setup, gVar has new value.
  A spec contains 2 expectations.

  nested describe
    gVar is global scope, tempVar is this describe scope.
```

与其他工具的集成

Karma

在 Java 中，用 JUnit 做单元测试, 用 Maven 进行自动化单元测试；

同样相对应的 JS 中，则可以用 Jasmine 做单元测试，用 Karma 自动化完成单元测试。

Karma 作为 JavaScript 测试执行过程管理工具，可用于测试所有主流 Web 浏览器。下面简单介绍一下 Karma 与 Jasmine 的集成。

首先，下载安装 Karma。

初始化 karma 配置文件 karma.conf.js。

安装集成包 karma-jasmine。

修改 karma.conf.js 配置文件。

需要修改：files 和 exclude 变量。其中 autoWatch 设置为 true，这样如果修改测试文件并保存后，Karma 会检测到然后自动执行。

```
module.exports = function (config) {
    config.set({
        basePath: '',
        frameworks: ['jasmine'],
        files: ['*.js'],
        exclude: ['karma.conf.js'],
        reporters: ['progress'],
        port: 9876,
        colors: true,
        logLevel: config.LOG_INFO,
        autoWatch: true,
        browsers: ['Chrome'],
        captureTimeout: 60000,
        singleRun: false
    });
};
```

启动 karma，自动执行单元测试。

```
F:\Projects\karma>karma start karma.conf.js
```

另外，Jasmine 也可以与持续集成工具 Jenkins 进行集成。

一个 Jasmine 的完整例子

Jasmine 在 JavaScript 中编译，必须被包含在一个 JS 环境中，比如一个 web 网页，来运行。JavaScript 被包含，通过一个<script>标签，然后所有以上的 specs 可以通过 Jasmine 计算和记录。这样 Jasmine 可以运行所有这些 specs。此页面被认为是一个"runner"运行者。

下面通过一个具体的例子来介绍通过 runner 怎样执行 Jasmine 的 suite。

首先，创建 HTMLReporter，Jasmine 调用它，来提供每个 Spec 和 Suite 的结果。

报告负责展现结果给用户。

代理为报告过滤 specs。允许点击某个结果中的 suites 或者 specs 来只运行 suite 的子集合。

当页面完成加载时运行所有的测试-然后确认运行任何之前的 onload 句柄。

见下面清单 9，完整代码示例见附件。

清单 9 .代码示例

```
<script type="text/javascript">
var jasmineEnv = jasmine.getEnv();
        var htmlReporter = new jasmine.HtmlReporter();
        jasmineEnv.addReporter(htmlReporter);
jasmineEnv.specFilter = function(spec) {
        return htmlReporter.specFilter(spec);
        };
```

```
window.onload = function() {
        jasmineEnv.execute();
        };
</script>
```

完整例子可以在附件中的 Reporter.html 运行，结果见下图：

图 3.测试用例结果

结语

通过本文的介绍，我们可以了解 Jasmine 的一些基本概念和用法，为组织项目的测试打下基础，为项目代码的可靠性和稳定性提供保证，并介绍了 Jasmine 和其他框架的集成。Jasmine 的一些相对高级的用法和技巧，会在后续的文章中进行介绍。

原文链接:
http://www.ibm.com/developerworks/cn/web/1404_changwz_jasmine/index.html?ca=drs-

Web Components - 面向未来的组件标准

作者:miller

首先需要说明的是这不是一篇 Web Components 的科普文章，如果对此了解不多推荐先读《A Guide to Web Components》。有句古话-“授人以鱼，不如授人以渔”，如果把组件比作“鱼”的话，对于前端开发者而言，W3C组织制定的HTML标准以及浏览器厂商的实现都是“鱼”而不是“渔”，开发者在需求无法满足的情况下通过现有技术创造了各种组件，虽然短期满足了需求但是由于严重缺乏标准，导致同一个组件有成千上万的相似实现但它们却无法相互重用，这很大程度上制约了组件化的最大价值-重用，Web Components则在组件标准化方面向前迈了一大步。

现状与困境

组件化给前端开发带来了极大的效率提升，组件化的UI框架也因此层出不穷，从EXTJs、YUI到 jQuery UI ，再到 Bootstrap、React、Ratchet、Ionic等等等等等等，几乎每年都有很多新的UI框架冒出来，它们或者借鉴或者颠覆其他已存在的框架。简单对比一下就会发现这些框架的很大一部分模块在功能上是重合的，但也仅仅在功能层面重合，代码层面确完全不兼容。

接下来选择 jQuery UI、KendoUI 以及 Bootstrap 中的Dialog组件从初始化、方法调用以及事件响应方面进行简单的对比。

jQuery UI

```
// 初始化
$( "#dialog" ).dialog({
  dialogClass: "no-close"
```

```
});

// 显示
$( ".selector" ).dialog({ show: { effect: "blind", duration: 800 } });

// 关闭事件
$( ".selector" ).on( "dialogclose",  function (e, ui) {
  // do something...
});
```

Kendo UI

```
// 初始化
$("#dialog").kendoWindow({
  actions: [ "Minimize", "Maximize" ]
});

// 显示
var dialog = $("#dialog").data("kendoWindow");
dialog.open();

// 关闭事件
var dialog = $("#dialog").data("kendoWindow");
dialog.bind("close",  function (e) {
  // do something...
```

```
});
```

Bootstrap

```
// 初始化
$('#myModal').modal({
    keyboard: false
});

// 显示
$('#myModal').modal('show');

// 关闭事件
$('#myModal').on('hidden.bs.modal', function (e) {
  // do something...
});
```

简单对比可以发现，几乎完全相同的功能在接口层面完全不兼容，导致使用者从某个实现切换到另一个实现时需要非常高的成本，这就是目前Web组件化方面无序和缺乏标准的一个写照。

再来看目前浏览器“内置”组件的现状，由标准化组织建立 HTML4、HTML5 等各种标准，浏览器厂商按照标准实现“内置”组件并声称兼容某某标准，开发者遵循标准来使用组件，使得代码可以在不同的浏览器里通过相同的方式来使用组件。

以“内置”组件video来简单示例：

```
// 初始化（直接写<video>标签或者通过javascript创建）
var video = document.createElement('video');
```

```
// 播放
video.play();


// 播放事件
video.addEventListener("play", function () {
   // do something...
 }, false);
```

相比使用各种组件框架来说，“内置”组件也是由不同的开发者（浏览器厂商）开发，但是由于遵循了相同的标准使得“内置”组件的使用在跨浏览器方面的成本大幅降低。

综上所述，组件框架目前无序、缺乏标准以及低效复用方面的问题需要通过组件标准化来解决，而Web Compo-nents则是标准化的一个很好的选择。

面向未来的组件标准

Web Components 的出现给组件标准化带来了很好的契机：

- WEB组件目前仍然依靠各种类似"Hack"的方式来模拟，模拟方式也各有不同，很难统一和标准化，而 Web Components 则直接提供了标准化的组件定义方式，这是组件标准化的基石，使得未来的组件能够统一创建、方法调用、事件监听、属性访问等。

- 基于标准化的组件定义方式，我们便可以像W3C等标准组织一样来定义组件标准，无需再依赖、等待“内置”组件，这也使得我们获得了“渔”的能力。

以上述的例子为例，未来可能会有一小撮人成立某个组件标准化组织-X，X的职责就是根据WEB组件的使用现状以及潜在的新需求来规范一个组件，包括组件的名称、方法、属性、事件。

例如《Dialog规范1.0》

- 组件名：x-dialog
- 属性：title
- 方法：show hide
- 事件：hide show

随后出现的UI框架宣称支持《Dialog规范》，但在实现上完全没有制约，可以是完全不同的实现方式、或者更好的性能、更炫的UI，而对于开发者而言，只需要写如下代码即可：

```
// 初始化(<x-dialog/>或者如下代码)
var dialog = document.createElement('x-dialog');


// 获取和设置title
var title = dialog.title;
dialog.title = title + '-_-';


// 显示
dialog.show();


// 关闭事件
dialog.addEventListener('hide', function( e ) {
    // do something...
}, false);
```

当用户不满意某个 Dialog 的实现而需要切换到其他实现版本时只需要引入不同的实现库，而不再需要重构代码。

```
// bootstrap
<link rel="import" href="/components/bootstrap/dialog.html">
```

```
// jQuery UI
<link rel="import" href="/components/jqueryui/dialog.html">

// Kendo UI
<link rel="import" href="/components/kendoui/dialog.html">
```

跨端的组件标准

集鹄在跨端组件实践 - 移动时代的前端一文中提到了跨端组件的概念。

跨端组件的实现同样面临着标准化的问题，Web Components 的标准化只规范接口，而底层的实现是完全自由的，自由到你可以使用 Web 技术来实现也可以使用 Native技术。

同样以 Dialog 为例，开发者可以在 Android 中用 Java 或者在 iOS 中用 Objective C 来开发声称兼容 《Dialog规范1.0》的组件，此时，Web 开发者的那段调用 Dialog 的代码不仅仅在 浏览器环境有效，在 Native 依然有效，而且调用的是 Native 实现，能够获得更为出色的性能。

总结

回顾浏览器的发展历史，也曾经历混乱和无序，随着W3C标准化组织的出现这一局面有了翻天覆地的变化，而对于Web组件而言，Web Components 的出现才仅仅是这一变化的开始，随着更为复杂的多端环境的出现，组件标准化还有着更大的想象空间。

作者：miller (http://milleris.me/) - The best preparation for tomorrow is doing your best today.

原文链接:
http://fex.baidu.com/blog/2014/05/web-components-future-oriented/

专访孔德芳：如何才能提高Java Web性能?

作者:张勇

摘要：孔德芳是Arcsoft视频流服务产品技术经理，同时也是CSDN Java Web版主，他拥有多年Java Web开发经历，并专注高性能Web服务、Web项目架构设计。CSDN记者近日就设计模式、如何提高Java Web性能等话题，对他进行了专访。

孔德芳认为，设计模式不会随着各种语言或开发框架的没落而没落，它犹如老酒，愈久愈醇香

孔德芳，Arcsoft云服务部门视频流服务产品技术经理。多年一线Java Web开发经历，专注高性能Web服务、Web项目架构设计。熟悉权限设

计、服务集群、安全防御、设计模式、多线程并发编程、JVM内存管理、项目管理等Web开发所涉及领域以及各种开源Web开源框架，还做过一年JavaMobile开发，熟悉多个J2ME开源框架。

CSDN：你对开源非常推崇，能说说其中原因吗?

孔德芳：中医领先西医上千年，到如今反而濒临一个被西医替代的尴尬境地，为什么？并不是中医无用，而是中医缺乏开源精神。中医讲究“祖传”、“独门”、“秘方”。中医一脉相承，西医却能集合全世界开源人士之力，大步向前迈进。开源！这样我们才能走得更快、更远。

当然，目前个人分享出来的只是一些小项目（见我上传的CSDN资源），有的只是跟随博客写一些小的示例性源码。如果社区有Java Web相关开源项目要开发的话，个人很乐意贡献自己的一点力量。

技术交流让我更优秀

CSDN：你在CSDN论坛担任Java Web版主已有一段时间，能不能谈谈这段时间都有什么收获？除了收获之外，有没有给你带来什么烦恼?

孔德芳：记得大学毕业时，有位老师对我们说，你在大学里学到的知识，工作以后能用上20%就很不错了。客观来讲，每一个人所做的工作所限，加上公司对员工的KPI是以结果为导向，我们不可能接触到我们所在领域的方方面面。这样对于我们大部分人来讲，我们积累到的东西，或许永远只是那么一小块，尤其是在一家公司呆的时间很长的同学，往往是深度有余而宽度不足。

通过在同一领域的技术论坛交流，我不仅能够了解并学习到我们工作中根本接触不到的知识，而且还能够认识到各种做法的优势和不足，进而和我们手头的方案进行对比。也就是说，通过对别人所遇到问题的分析和解答，我额外积累到了别人的一些工作经验，而且通过各位回复者的意见，我还能总结到找到某种情况下的最佳解决方案。再者通过参与大家的交流，也能随时把握当前应用最广泛的最主流框架技术，避免自己的团队与别人脱节，甚至闭门造车。

还有一些是自己工作中所遇到过的问题，有人提问，可以直接给出解答。随着参与进来的网友增加，我还能借鉴到其他朋友的做法，进而对我们工作中的一些做法进行改进。比如有一次，有网友问到了关于Maven的两个问题：一个是选择性拷贝另一个是jar依赖管理。我们的项目管理用的刚好是Maven，我给楼主提供了我们的解决方案。楼主表示感谢的同时又提了一个配置文件分离的问题，由于我们在项目中也是手工分离的，所以没有办法帮楼主解决这个问题。后来，另一个网友也参与了进来，他提供了他们的做法，而这正是这个问题的解决办法。这下好了，我们以后的各个测试环境和生产环境的部署中再也不用手工去调整配置了。

另外一些是分享自己面试或者工作经历。这些都是网友们自己在工作学习中所总结出来的心得体会，具有一手的参考价值，甚至有的可以直接拿来为我所用。这些信息都具有极大的实战性和保险性，是我们去看参考书和API所找不到的资料，对于我来讲都是很宝贵的。

还有这段时间我也认识了很多CSDN的网友，包括Java版的另外几位版主，大家在论坛里进行技术讨论，有很大的技术互补作用。另外我们还通过QQ交流，在个人工作经验甚至生活上进行一些沟通，有的甚至有一些私活介绍给对方的团队……这些都是同一个领域内的人脉积累，是除了同事、同学、朋友之外的一个积累，扩大了我的圈子。

烦恼嘛，还真有一些。比如很多网友在提问之前不知道先在坛子里搜索一把，其实他提问的问题早不知道别人提过了多少遍，造成很多“月经帖”甚至“日经帖”。面对这种帖子，时间久了，总有一些做重复无谓劳动的感觉。另外一些网友只管问，不管结帖，这是对回答者付出的不尊重，某种意义上打击了回答者的积极性。还有一些网友缺乏分享精神，发了个问题，问题解决了就结帖，说一声问题解决了，却不公布解决办法，很多遇到同样问题搜索到这个页面的朋友是带着希望而来，而又抱着失望而去。

不断学习是软件行业弊端?

CSDN：有人讨厌软件行业，他们觉得这个行业知识更新速度太快，得不断学习才能跟上时代步伐，对于这种心态你怎么看?

孔德芳：从我开始写程序，就在各个论坛见到一种热点讨论，大概就是程序员到底是不是终身职业，35岁是不是程序员的退休年龄。还有一些行业外的朋友，碰到我就很关心地问，听说软件开发是吃青春饭的，等你以后年纪大了怎么办？一时间弄的人心惶惶，人人自危，很多人抱怨入错了行。

其实每个行业都有各自的弊端。刚才还有个体制内的朋友给我抱怨："上周末我在办公室写战略报告就觉得好委屈，就那点死工资，还要这么努力工作，好不甘心"。我安慰他，我们想进去，还找不到门路呢。其实这是大家的习惯性思维，很容易把周围的人的缺点，或者自己所从事行业的弊端进行无限放大。我们来做一个实验：请给我一张大的白纸。我在白纸的中央画一个黑点，然后展示给大家看，问大家看到了什么？大部分人的答案应该是黑点。我那么大一张清清白白的白纸，你没有看到，就中间那么一个小小的污点，就成了你的标准答案。我们不要对周围的人过分要求，把我们所从事行业的弊端无限放大。

软件行业的弊端是什么？在我看来，这个行业的真正的弊端就是，它现在还是一个新兴行业，很多事情没有一个量化的标准。这样带来的害处就是，没完没了的加班，工作强度的加大，这对程序员的身体是一种摧残。这种现象很普遍，事实情况是，很多程序员正是受不了这种摧残而离开这个行业。无限的加班，有些软件公司还是单周末，好容易歇个周末，过度疲惫的程序员选择在家里睡一天。这样玩法，铁打的身子也扛不住啊。真心期望，也许有一天，软件开发能够像其他传统行业一样，有一个量化标准，程序员们不再加班。在这一天到来之前，**我忠告各位程序员同行：除非你真的做好了35岁就退休的打算，否则的话，珍爱身体，远离加班**。

这里，有的同学要说了，你跑题了，问的问题是**程序员得不断学习是软件行业的弊端。——我可不认为知识更新速度太快是软件行业的弊端，能够不断学习是好事。如果一个岗位，经过短短数月的培训就可以一辈子不用学习，假以时日，他站在整天需要学习的其他岗位的工作者面前，就是一个懵懂的小孩站在一个白发苍苍的老者面前，是愚者与智者的比较了，也是打酱油与专业的比较**。在职场上真正经得起风雨的人，是那些有真才实学的人，有"空杯心态"的人。人们有的时候认为自己在某个行业里做了很多年，就认为是这个行业里的行家里手，没有我不懂的东西。于是别人在自己眼里都是外行，别人讲的东西都听不进去，要知道"天外有天，人外有人！"。在知识经济时代，科技飞速发展，知识更新加快，如果不虚心学习新的知识和方

法，即使你原来的专业知识很扎实，也一样会被社会的进步潮流所淘汰，所以要活到老，学到老。只有定期给自己复位归零，清除心灵的污染，才能更好地享受工作与生活。我认识很多40多岁的同事，在CSDN上也有一些40多岁的程序员，他们在各自所在领域是当之无愧的专家。通过跟他们的沟通，我发现他们都有一个共同特点：每当实现了一个近期目标，决不自满，而去迎接新的挑战，把原来的成功当成是新成功的起点，树立新的目标。正是他们都勇于打破瓶颈，不断实现自我突破，才使得他们走在了时代的前列。

设计模式犹如老酒，愈久愈醇香

CSDN：你对设计模式也有所研究，能不能谈下如何在实际项目中应用设计模式?

孔德芳：编程语言都在编程思想上是相通的，设计模式是软件工程的基石。伟大的Java缔造者们将设计模式的应用发挥到了极致，作为解释型语言的Java从诞生到今天，始终能够作为最主流与应用最广泛的语言力压其他众多的开发语言，与缔造者们不遗余力地提高其健壮性、高性能是分不开的，而设计模式在其中无疑起了举足轻重的作用。如果你已经疲惫于层出不穷的开发语言，如果你已经眼花缭乱于日新月异的开发框架，那么就学设计模式吧，**设计模式不会随着各种语言或者开发框架的没落而随之没落，甚至愈如浓醇的老酒，愈发散发出迷人的醇香。设计模式是前辈们的一种经验共享的方式，或者说是一套内功心法**。前人在修炼内功的时候就已经考虑到我们这些后来人了，总结了这么一套内功心法，避免了我们重复走他们走过的错路、弯路。

首先是要学习，工作再忙，也要坚持学习设计模式，不然就真的成了只会搬砖的码奴了。

然后是注意观察。学习了一种设计模式，要时时刻刻心里都有一个魂，看看我们接触到的软件里，哪里应用了这种设计模式？比如我学习观察者模式之后，再去学习 MINA框架，发现MINA的事件处理机制就是运用了观察者模式；还有我在Wowza插件开发的时候，发现Wowza的插件模块扩展开发整个就是一个观察者模式的典型，用户通过自定义模块对比如com.wowza.wms.stream.IMediaStreamActionNotify等接口的实现，可以捕

捉到自己所关心流的一系列事件，进而就可以对特定直播/
点播频道进行监控了。

接下来就是模仿。学习概念不是目的，我们应该在实际项目中切实感受设计模式带来的好处，领略设计模式的真正的威力，而不只是用来玩理论、侃大山。我们在实际项目的设计中应该时时考虑对比，设计模式能否给我们提供更优方案？一定要多用，不用的话，我们永远不能把这些内功心法和自己的主修功法融为一体。一开始可能只是模仿，但别怕，多思考多用，同一个模式用上几次之后，我们就能够对其适用场景以及优缺点有个自己的评判。下次再遇到类似问题，我们可能就已经能够利用设计模式归结出数个解决方案，并最终有所取舍得出最佳方案。

比如我在经过学习过观察者模式，并模仿前人软件中的应用之后，在一个类似上海抢拍车牌号的场景下，一下子就想到了观察者模式。这种情况下一条线程不断地对当前价格进行刷新，同时几十条线程（几百个对象）对当前价格进行读取监控。用观察者模式效率比较好，可以解决由于线程竞争、加锁而带来的效率问题。把读数据的线程归为观察者，主题是缓冲区数据。一旦数据有更新，主题向观察者推送更新数据，这样推数据的做法效率很高。缓冲区做成主题，每个观察者都有一份自己关心的主题数据本地备份，如果主题没有推数据过来，本地备份就是最新数据。当然，这么干消耗空间，但是却换得多线程环境中效率上的大幅度提升，这就是所谓的用空间换时间。显然，这个时候，空间不是瓶颈，而程序的执行效率，客户能够拿到最新报价的时间才是我们最关心的。

CSDN：从你CSDN博客上获知，平时你翻译了不少技术文档，为什么翻译这些文档？另外，你平时还有些什么好的学习习惯？

孔德芳：其实这些文档也是我正在学习的内容。比如那个MINA2.0用户指南，其时我们在做一个视频服务项目，刚好用了MINA，API没有中文版的，而且关于MINA的中文资料和文档也很少，能够系统介绍这一块的更是难以查找。所以，我在查阅官方资料的同时，顺手也翻译了一把，并整理成一本电子书，以便自己查阅，也希望可以方便到其他对MINA感兴趣的同学。

一些好习惯有：技术学习，就是积极参与同行业技术讨论，积极参与各种社区组织的技术分享交流大会，多结交一些同一领域的朋友。好记忆不如

烂笔头，一定要把各种要点记录下来，坚持写日记，几年下来，我的日记本写满了5、6本；技术积累，坚持写博客，记录自己技术心得；技术实践，多年一线Java Web开发经验，主要是企业管理、企业办公、网站、电子商务等方面的项目，也做过一年javamobile开发经历，主要是通用低端手机平台移植性高的项目(J2ME)，目前就职于一家公司的云产品部门，负责视频流服务产品的研发；项目开源，把不牵涉到公司业务方面的项目开源出来，不做保留。

你说对了，学习+积累+实践+开源，这就是我这些年积累到的技术之路心得体会。

Java是应用于大型网络应用的最好语言

CSDN：在众多领域中，为什么选择Java Web开发之路?

孔德芳：现在是互联网时代，国内外信息化建设已经进入基于Web应用为核心的阶段。Java作为应用于网络的最好语言，前景无限看好。最重要的是，Java开源，前面提到了，开源的环境之下我们才能走得更快、更远。

CSDN：做Java Web项目需要掌握哪些技术？其中，哪些技术是最基础、最重要的?

孔德芳：要做Java Web项目，需要掌握的技术有：Java语言、面向对象分析设计思想、设计模式和框架结构、XML语言、网页脚本语言、数据库、应用服务器和集成开发环境。

最基础最重要的就是Java语言、面向对象分析设计思想、设计模式和框架结构。Java语言体系比较庞大，包括多个模块。从WEB项目应用角度讲有JSP、Servlet、JDBC、JavaBean（Application）四部分技术；Java语言是完全面向对象的语言，在分析项目业务关系的时候应用一些UML图能尽快找出业务逻辑主要面对的对象，然后对每个对象进行行为划分，最后再实现对象之间的集成和通信；设计模式主要在与两层的设计模式、三层的设计模式和N层的设计模式，很多的Web项目采用的是MVC的三层开发结构，也就是JSP+Servlet+JavaBean。

CSDN：Java Web优势有哪些？主要适用于哪些类型项目？

孔德芳：Java Web的优势体现在：

1. Java语言是一门不会“死掉”的语言。全球有成千上万的Java开发者，据CSDN软件开发者2013年的调查显示，Java背后的开发者比例占有高达45.39%；

2. 众多的开发者，意味着要找到一个程序员来对现有系统进行维护是相当容易的一件事情（这对公司来讲是个好消息）；

3. JAVA/J2EE 体系的强大和优雅使得我们可以精心去构建一个良好的系统；

4. 众多的免费工具，比如Apache/Tomcat/JBoss， 这些都是构建一个网络程序的坚实基础；

5. 为开发人员提供的优秀的支持。比如Eclipse，Ant，Maven；

6. 众多的JAVA/J2EE 核心库以外的第三方库使得我们开发附加功能轻而易举；

7. 巨头商业供应商提供的工具支持，比如Oracle，IBM/Rational 等等；

8. 语言结构更新版的持续研发。

对于比较简单的小型应用，Java Web并没有多大优势，比如对于只有一个数据库、简单几个表结构的博客网站，完全可以用其他更简单的方案搞定。Java Web适用于需要和其他一些系统进行交互的大型Web应用，比如银行或者保险公司的项目。

CSDN：能不能和大家分享下常用的Java Web框架？其中哪些是你常用的？为什么？

孔德芳：目前比较流行的Java Web框架有：Struts1/2、hibernate、spring、iBatis、MyBatis等等。我用的比较多的是spring和ibatis，它们功能丰富，漏洞少，扩展性好，相当灵活，上线项目性能也不错，而且团队人员都比较熟悉，用的都很顺手。

CSDN：使用开源框架有哪些利和弊？开源框架对Java Web技术发展有哪些影响?

孔德芳：

利：

1. 免费；
2. 稳定、安全（相对）、容易找到各种层次的技术人员；
3. 技术人员上手快；
4. 可以在需要时候深入了解架构设计实现原理，可根据自己的需求进行定制，也可借鉴其设计思想并引入到自己的应用中；
5. 用的人多，出了问题很容易找到解决方案，而且与人技术沟通也方便；
6. 使用开源框架的成功案例多，性能等各方面有保障。

弊：

1. 没有商业支持；
2. 随处可见的框架屏蔽了各种技术细节，多数技术人员知其然不知其所以然，使得其很职业发展路线中很容易遇到瓶颈。

如何才能提高Java Web性能?

CSDN：如何提高Java Web项目的性能？都是从哪些方面来改进?

孔德芳：关于Java Web项目的性能，每个人所面临的情况都是不一样的，多跟踪，多思考，多尝试。以下是我总结的一些思路，仅供参考吧：

1.使用Nginx作为前端接入

用Nginx进行动静分离。这个不用多讲，新浪、网易、淘宝、腾讯等巨头的使用已经说明了一切。

2.保持最简单的架构

遵守KISS原则（Keepitsimpleandstupid）。尽量不要考虑项目外的重用，过多的考虑项目外的重用，必然会增加项目的复杂度。避免过度集成，让每个模块只做自己的事，这对于日后的维护和模块复用都有好处。

3.精心设计缓存处理、毫不吝啬代码（对象、列表、片段）

对于门户网站的首页来说，往往可能会有近百个SQL。用户并发上去以后，光首页就足以让服务器down掉。缓存不但有利于降低负载，而且还能提高响应速度。

4.调整使用聚集索引

对于每个表来讲，聚集索引只有一个，利用好了，查询速度会有意想不到的提升效果。

以MySql为例，InnoDB选取聚集索引参照列的顺序是

1）如果声声明了主键（primarykey），则这个列会被做为聚集索引;

2）如果没有声明主键，则会用一个唯一且不为空的索引列做为主键，成为此表的聚集索引;

3）上面二个条件都不满足，InnoDB会自己产生一个虚拟的聚集索引。

[java] view plaincopy

```java
CREATETABLE`timeline_raw`(

`rawId`bigint(20)NOTNULLAUTO_INCREMENT,

`uid`bigint(20)DEFAULTNULL,

`did`bigint(20)DEFAULTNULL,

`channelId`char(1)NOTNULLDEFAULT'1'COMMENT'1:qvga;2:720p',

`fileId`bigint(20)DEFAULTNULL,

`sectionId`bigint(20)DEFAULTNULL,

`headerFilePath`varchar(120)DEFAULTNULL,

`startTime`bigint(20)DEFAULTNULL,

`endTime`bigint(20)DEFAULTNULL,

`updateTime`datetimeDEFAULTNULL,

`createTime`datetimeDEFAULTNULL,

PRIMARYKEY(`rawId`),

KEY`index_uid_did_startTime`(`uid`,`did`,`startTime`)USINGBTREE,

KEY`index_uid_did_endTime`(`uid`,`did`,`endTime`)USINGBTREE,

KEY`index_time`(`startTime`)USINGBTREE,

KEY`index_uid_did_fileId`(`uid`,`did`,`sectionId`)USINGBTREE,

KEY`index_sectionId`(`sectionId`)

)ENGINE=InnoDBAUTO_INCREMENT=1DEFAULTCHARSET=utf8
```

这个表有四个索引：主键rawId、sectionId、`uid`,`did`、startTime。

项目的iBatis2中有这样一条查询语句:

```java
<selectid="getRawFileList"parameterClass="java.util.HashMap"resultClass="com.defonds

SELECT*FROMtimeline_raw_

WHEREuid=#uid#

ANDdid=#did#

ANDchannelId=#channelId#

<isNotNullproperty="sectionId">ANDsectionId=#sectionId#</isNotNull>

AND

(

(startTimeBETWEEN#startTime#and#endTime#)

OR

(endTimeBETWEEN#startTime#and#endTime#)

OR

(

<![CDATA[

startTime<=#startTime#

]]>

AND

<![CDATA[

endTime>=#endTime#

]]>

)

)

ORDERBYstartTime;
```

```
46.
47. </select>
```

根据实际业务向timeline_raw表注入一千万条数据，进行模拟测试，发现getRawFileList的执行平均时间为160ms以上。这是不能接受的。

考虑到实际业务中对于主键rawId查询条件甚少，我们把rawId主键索引取消掉，改为唯一约束，却把sectionId+startTime+endTime作为主键（业务上能够保证其唯一性，根据InnoDB索引规则，这个索引将成为我们新表的聚集索引）。然后把sectionId、startTime两个索引也取消掉，仅保留`uid`,`did`索引。

这样子，我们新表的索引实际上只有两个了：一个聚集索引(sectionId+startTime+endTime)一个非聚集索引(`uid`,`did`)。

再次进行模拟测试，同样的数据、数据量，同样的查询结果集，getRawFileList执行平均时间已经降到了11ms。结果是令人振奋的，不是么?

5.使用/dev/shm来存储缓存的磁盘文件

在网站运维中，利用好了这一点，往往有意想不到的收获。以tomcat为例，可以通过修改catalina.sh中的CATALINA_TMPDIR值的路径来将缓存设置为/dev/shm。

以OSC为例，他们就是纯Java写的，部署在tomcat下。在长时间的在线运行之后，管理员发现网站响应速度奇慢，服务器负载正常，又找不出是哪里的问题。后来df一下，发现tomcat临时目录下的文件足足有8G之多，原来是CPU等待磁盘操作造成响应速度加长。于是他们将临时目录映射到/dev/shm，网站响应速度从此奇快。

6.分析系统中每一个SQL的执行效率

以MySql为例，对于每个SQL最好都explain一下。对于有明显效率问题的，通过sql优化、调索引等方法进行改进。

7.健康慢查询日志，检查所有执行超过100毫秒的SQL

对于上线了的项目，健康慢查询日志，检查所有执行超过100毫秒的SQL，看看有没有优化余地。对于没有上线的项目，可以进行场景模拟对嫌疑SQL，或者对频繁使用的SQL进行性能测试，统计它们执行时间，得出平均值，画出曲线分析图，对于单表千万数据，执行时间超过50ms的SQL要重点关注。

CSDN：对于一些规模较大的Java Web项目，都是如何处理的？比如分工等等？另外，在开发效率上有没有什么心得？

孔德芳：对于一些规模较大的Java Web项目，要完成工作任务，仅仅靠努力工作是远远不够的。首先要明确作战目标：做一个立项报告，让团队明白项目的目标，否则大家都是无头的苍蝇；然后是分工明确：和大家在一起分析各个人的工作情况，明确各个人的职能和职责；之后是预算：我喜欢让每个人按照自己的任务先进行自己预算，如果和自己给他估算的相差很大再去仔细分析；最后就是跟踪或者敦促执行情况了。具体项目分工我推荐阿朱的四套马车。

在提高开发效率上，我的做法是这样的：指导并直接参与核心模块研发、跟踪组员任务进度：监督组员完成情况和质量，定期组织团队codereview、对完成情况不太好的组员，或者所在任务技术难度比较大的组员提供技术支持和帮助。

CSDN：在Java Web项目架构设计上有没有什么经验可以分享？

孔德芳：

1. 拆分（软件部署方式（业务、架构（展现层、业务逻辑层、持久层））） 软件包括自己的、第三方的（DB、MQ...）；
2. 抽取通用服务，形成自己应用的中间线；
3. 负载均衡（LVS、HProxy、读写分离、切片）；
4. 高可用\故障转移（heartbeat、keepalive、MasterSlave）；
5. 共享存储（nfs、iscsi）；

6. 在业务和技术之间做出最佳的选择。

CSDN：如今云计算应用非常广泛，Java Web能否适应云计算发展需要？在开发上，Java Web如何与云计算相结合?

孔德芳：云计算按照服务类型大致可以分为三类：基础设施即服务IaaS、平台即服务PaaS、软件即服务SaaS。

IaaS：AmazonEC2/S3、Openstack、Cloudstack；

PaaS：GoogleAppEngine、MicrosoftWindowsAzure、Openshift；

SaaS：AmazonDevPay。

云计算另一种划分方式：桌面云、公有云、私有云和混合云。

支持云计算的几种关键技术：（google三论文）

分布式文件系统：GFS（google）、HDFS（hadoop）、GridFS（MongoDB）；

分布式并行计算框架：MapReduce(google)、MapReduce(Hadoop)、MapReduce(MongoDB)、EMR（Amazon）；

NoSQL：BigTable（google）、Hbase（hadoop）、MongoDB。

（虚拟化）

Xen；

KVM；

VirtualBox；

VMWareExsi。

传统应用按照数据处理分为OLTP、O-LAP两种，按照部署方式分为CS、BS（Java Web...），BS架构的应用可以在合适的时候引入云计算的各种技术，以获得更高的可靠性（failover）、性能（horizontalshard、verticalshard）、扩展性（scalein、scaleout）。

视频流服务上的一些分享

CSDN：你就职于某云产品部门，在视频流服务领域有没有什么心得分享?

孔德芳：目前能够提供视频流服务的服务器有很多，比如AdobeFMS、haXeVideo、RealNetworks的HelixUniversalServer、Red5MediaServer、Erlyvideo、UnrealMediaServer、WowzaMediaServer、WebORBIntegrationServer以及NginxwithRTMPModule等等。选择一款适合自己的流媒体服务器很重要，比如，AdobeFMS服务性能都很稳定，但不开源而且收费昂贵；Wowza服务性能也很好，可扩展性很高，技术支持不错，但不开源，价格相对实惠，性价比高；Red5服务性能也都不错，而且开源，但自2012年下半年起官方停止任何技术支持，而且不再有新版本发布；NginxwithRTMPModule开源，而且技术支持也不错，但有很多bug，而且并发性也不太好。

CSDN：有人说，初学Java Web开发的人，要远离各种框架，对此，你怎么看?

孔德芳：我支持这种说法。框架把程序员要做的很多事情封装起来，让我们能够专注于企业业务开发，能显著提高开发效率。但是一开始应该注重基础和原理，初学Java Web开发，一开始就上手框架，很容易忽略掉那些基础知识。很多人认为Java Web开发就是S-SH，一系列的配置文件拷来拷去；很多程序员做了几年Java Web，你问他定义的一个对象的生命周期描述一下，以及为什么要用spring管理，他都答不出来；还有少数一部分人更是连js、jsp哪个是在服务器端执行哪个是在客户端执行都分不清楚，弄出来很多本来期望在客户端弹出的窗口却在服务器端弹出来的笑话…这些都是一开始学习上手框架，没有注重基础学习的后果。可以说成也框架，败也框架。

学习Java不能本末倒置：应基础后框架

CSDN：你觉得该如何系统学习Java Web？有没有什么学习窍门？

孔德芳：我觉得要系统学习Java Web开发，一定要抓住这两样，JavaServlet规范和HTTP协议。Servlet规范去官方看文档，HTTP协议推荐图灵的那本《HTTP权威指南》。这些没问题了，再去学框架。框架只是工具，这些是基础，不要本末倒置。

CSDN：在Java Web学习上，有没有什么学习难点？如何解决这些难点？

孔德芳：**难点是Web的MVC实现、项目的测试和调试**。项目的测试和调试是学习难点，这个没得说，这是每个程序员的基本功底，从某种意义上来讲能够代表一个程序员水平的高低，需要我们在项目实践中多动手、多思考，不断自我积累和提高。**那么MVC这种大家耳熟能详的东东，怎么也成了难点了？事实上，我经常在Java Web版看到网友把SSH进行MVC的三层对号入座。这就暴露出来他根本就没有理解MVC**。事实上，**MVC的三层和所谓SSH不是一个概念，不要硬性按号入座**。SSH体系的分工是为纵向分工是线性分布，而MVC是面性的三者之间都能互相合作，虽然ssh目的和MVC一样，都是支持可维护性可扩展性为目的。

CSDN：Java Web程序员在熟练使用一些框架之后，如何进一步提高技术？

孔德芳：Java Web程序员在熟练使用一些框架之后，要想避免成为熟练工，突破现有瓶颈，应该多关注以下这些问题：

1. 海量数据的处理
2. 数据并发的处理
3. 文件存贮
4. 数据关系的处理
5. 数据索引
6. 分布式处理
7. 安全防御

8. 数据同步和集群的处理

结束语：我的程序员之路是从CSDN开始

孔德芳在最后聊到，他的程序员之路就是从CSDN开始的。

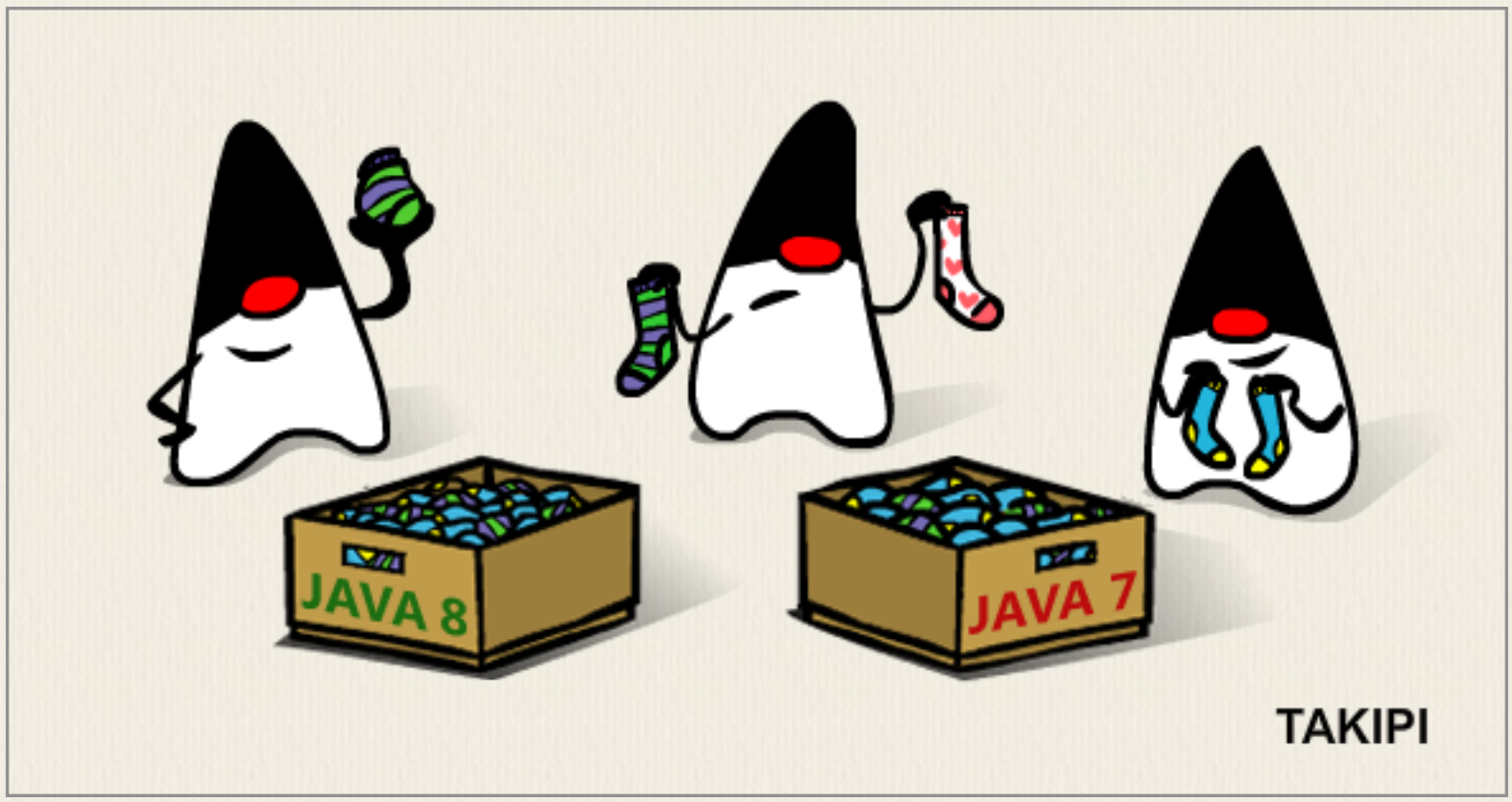
“大学那会，一个偶然的机会，接触到了CSDN，CSDN的人气的活跃度、各种技术讨论的广度和深度深深吸引了我，从此我开始频繁关注这个程序员聚集的大本营。当然，一开始大部分时候只是潜水，看别人的讨论、分享，还有查阅一些资料，因为自己所知道的不多，好像根本就参与不进各位的技术热点的讨论中去，最多也就是用早期申请的几个账号提问几个问题而已。直到2007年，注册了现在的这个账号，自己也参与论坛讨论、博客分享中去，竟然一发不可收拾。”

就此，孔德芳无比感慨地称，感谢CSDN的那些帮助过他的人，虽然有些问题似乎是微不足道的问题，但正是你们的帮助，让自己一次次从没有头绪中豁然开朗起来，也正是由于你们貌似很不经意的一次次指引，让他不至于迷失，一步步走过那开始时的懵懂。

原文链接:http://www.csdn.net/article/2014-04-28/2819531/1

Java 8的新并行API – 魅力与炫目背后

翻译:ImportNew-zhaoxin



我很擅长同时处理多项任务。就算是在写这篇博客的此刻，我仍然在为昨天在聚会上发表了一个让大家都感到诧异的评论而觉得尴尬。好吧，好消息是我并不孤单——Java 8在多任务处理方面同样很优秀。让我们来看看它是怎么做的。

在Java 8引入的新功能中，有很重要的一项是并行数组处理。这项新功能使得我们能够使用可以利用多核体系结构的Lambda表达式来对对数组的元素进行排序，过滤和分组。这里的重点是，Java程序员只需要非常少的工作就可以立刻使程序的性能获得提升。非常酷。

问题来了。这项新功能有多快？我应该什么时候使用它？好吧，答案有点让人沮丧——这依赖于具体的情况。要知道依赖什么情况吗？请继续阅读。

新的API

Java8的新并行操作API十分灵活。让我们一起看几个我们要用来做测试的例子。

1. 使用多核对数组进行排序：

```
Arrays.parallelSort(numbers);
```

2. 根据特定的条件（比如：素数和非素数）对数组进行分组：

```
Map<Boolean, List<Integer>> groupByPrimary = numbers
    .parallelStream().collect(Collectors.groupingBy(s -> Utility.isPrime(s)));
```

3. 对数组进行过滤：

```
Integer[]  prims = numbers.parallelStream().filter(s -> Utility.isPrime(s))
    .toArray();
```

跟自己写多线程程序来实现相同的功能比较，生产力提高太多了！在这个新的体系中，我个人最喜欢的是一个叫Spliterator的新概念，将一个集合分成多个块，并行处理这多个块并将处理结果汇合到一起。就像它的哥哥iterator，它也被用来遍历一个集合的元素，只不过它更加灵活，允许你编写检查和分离集合的自定义行为，并在遍历时直接插入。

它的性能如何?

为了测试这些并行操作API的性能， 我在两种情况（低竞争和高竞争）下进行了实验。原因是单独运行一个多核算法，往往会有好的性能，但在真实的服务器环境中运行，情况就完全不同了。真实环境中往往有大量的线程在竞争宝贵的CPU时间片以处理消息或用户请求，由于竞争的存在，程序的性能就降低了。所以我进行了接下来的测试。我首先随机生成了长度为100K的整数数组，这些整数的取值在0到1百万之间。然后我分别使用传统的顺序方法和新的Java 8的并行API对这个数组进行了排序，分组和过滤。结果并不使人惊讶。

- 快速排序快了4.7倍
- 分组快了5倍
- 过滤快了5.5倍

这可以说明java 8的并行API具有非常好的性能吗？很不幸，不能。

Sort (ms)	Parallel Sort	Group (ms)	Parallel group	Filter (ms)	Parallel filter
23	4	18	4	18	3
20	4	15	3	16	3
20	5	17	3	17	3
20	4	18	3	29	3
20	5	15	3	15	3
19	5	17	3	15	3
21	4	16	3	17	3
19	4	16	4	14	3
19	4	17	3	18	3
19	4	15	3	15	3
200	43	164	32	174	30

*测试结果与运行了100次的附加测试结果一致。
*测试机器为MBP，i7四核。

在有负载的情况下会发生什么呢?

目前为止新API的性能表现非常出色，原因是线程之间对CPU的时间片的竞争非常少。这是理想的环境，但不行的是，理想环境往往不会出现在现实环境中。为了模拟真实的环境，我建立了第二个测试。这次测试使用跟第一次相同的算法，但测试任务在十个并发线程上执行，以模拟处在压力环境中的服务器同时处理十个请求的情况。这十个请求使用传统的顺利处理方法或Java 8的新API处理。

测试结果

- 排序现在只快了20%
- 过滤现在只快了20%
- 分组现在满了15%

更高的规模和竞争水平很可能使这些数字进一步下降。原因是在一个多线程的环境中添加线程并不一定能帮助你提高计算效率，是计算机的CPU个数决定了计算效率，而不是线程个数。

Sort (ms)	Parallel Sort	Group (ms)	Parallel group	Filter (ms)	Parallel filter
36	29	19	26	28	29
39	27	22	48	30	29
33	38	25	24	30	21
37	27	25	24	30	24
39	35	23	24	30	22
39	34	27	24	32	22
37	37	30	25	31	21
30	33	25	22	30	20
30	37	22	37	24	19
31	35	22	37	26	18
351	332	240	291	291	225

结论

虽然这些都是非常强大和易于使用的API，但它们不是银弹。我们仍然需要花费精力去判断何时应该使用它们。如果你事先知道你会做多个处理并行操作，那么考虑使用排队架构，并使并发操作数和你的处理器数量相匹配可能是一个好主意。这里的难点在于运行时性能将依赖于实际的硬件体系结构和服务器所处的压力情况。你可能只有在压力测试或者生产环境中才能看到代码的运行时性能，使之成为一个“易编码，难调试”的经典案例。

原文链接： takipiblog 翻译： ImportNew.com - zhaoxin

译文链接： http://www.importnew.com/11113.html

比特币丢失、MongoDB和最终一致性

作者:Abel Avram 译者:马德奎

近日，多家比特币运营商失窃，这引发了一场争论，最终一致性数据库对银行业务是否有用。

2014年3月2日，由于代码缺陷，Flexcoin丢失了它所有的比特币。攻击者发出了成千上万的并发请求，定序将比特币从他其中一个账户转移到另一个账户。之后，他用其它账户重复同样的操作，直到取走了所有比特币。之所以能够这样做，是因为编写的代码没有处理多并发请求，而且所有转移都是发生在余额更新之前。如果余额没有及时更新，即使账户是空的，请求也可能被批准。因此，在丢失了896个比特币（价值约50万美元）之后，Flexcoin关停了他们的业务。

两天之后，Poloniex发生了同样的事，但他们只丢失了12.3%的比特币，而且该公司弥补了损失，并设法维持了下去。

康奈尔大学副教授Emin Gün Sirer写了一篇博文，将比特币丢失归因于最终一致性数据存储。在容易产生银行盗窃的NoSQL解决方案中，他提到MongoDB、Cassandra和Riak，因为：

这里的问题，其根源在于MongoDB提供的接口和语义设计有问题。如果我们用的是Cassandra或者Riak，那么情况不会有任何不同……

比特币恰逢分布式系统的一个尤其黑暗的时期，人们秉持对CAP理论的错误理解，认为他们只不过是不得不放弃数据库的一致性……

目前尚不清楚Flexcoin或者Poloniex那时是否正在使用MongoDB，而值得一提的是，Sirer正参与开发HyperDex，它支持ACID事务，是一个有竞争

性的键-值数据存储。另外，这不是Sirer第一次诟病MongoDB的设计了。一年前，他就声称MongoDB的容错性有问题。

抛开争论不谈，最终一致性数据存储是否适合银行业务是个值得深思的问题。不出所料，Sirer的博文引发了大量的评论，本文节选了部分最值得注意的。

jrp指出，更新操作可以使用MongoDB在数据库级以原子方式实现，但他也认为“这将照顾不到其它ACID属性。”

jakcharlton认为，鉴于最终一致性在这种情况下有问题，一个“ACID系统并不能解决该问题，它只是将问题推到应用程序的边界，问题在那里再次发生。银行业务使用最终一致性数据存储是个坏例子，也显示出开发人员思维方面的问题，他们认为由于他们的数据库满足ACID，所以他们能够免于并发问题。”

Alex“做任何事都使用MongoDB，除了需要事务的时候，那时我会用合适的工具（不是MongoDB）完成工作。”他认为，将MongoDB用于不该使用它的工作是开发人员犯的一项错误。

Robert Escriva是一名HyperDex开发人员，他认为罪魁祸首不是程序员，而是最终一致性系统可以用于银行业务这样一种普遍存在的观念：

问题不在程序员的理解。有一种普遍存在的错误观念鼓励人们使用最终一致性系统。“如果它好到足以用于银行，那么它也能满足你”，他们会用这样的话来证明它的合理性。这种想法是危险的。

最终，应用程序应该是其不变量的总和。如果系统底层的数据存储不能提供保证——或者更糟糕，需要大量的维护以及10万美元的支持合同来提供保证——应用程序有了问题，却归咎于开发人员，这是一种托词。我们应该以更高的标准要求数据库供应商（尤其是那些动辄就获得数亿VC现金的供应商）。

Eric Brewer是CAP理论的创建者，他先前在一篇文章中写道，为了在分区期间提供可用性，银行在他们的ATM业务中放弃了一致性。但他们这样做的时候采取了一定的防范措施，其中包括将取款数额限制在某个较小的阀值内。这里还要提一下Stripe，根据他们的一篇博文，这是一款使用了MongoDB的Web&移动支付系统。

最终一致性数据存储适合一般银行业务吗？或者开发人员应该知道它们的局限性而另寻方案呢？

原译文链接:

http://www.infoq.com/cn/news/2014/04/bitcoin-banking-mongodb

原文链接:

http://www.infoq.com/news/2014/04/bitcoin-banking-mongodb

我的算法学习之路

作者:Lucida,学生生涯在大工和北航度过,现就职于Google London;LuSoft和IdaSoft的创建者,其下包含拨号助手和金庸全集等热门应用

关于

严格来说，本文题目应该是我的数据结构和算法学习之路，但这个写法实在太绕口——况且CS中的算法往往暗指数据结构和算法（例如算法导论指的实际上是数据结构和算法导论），所以我认为本文题目是合理的。

这篇文章讲了什么?

- 我这些年学习数据结构和算法的总结。
- 一些不错的算法书籍和教程。
- 算法的重要性。

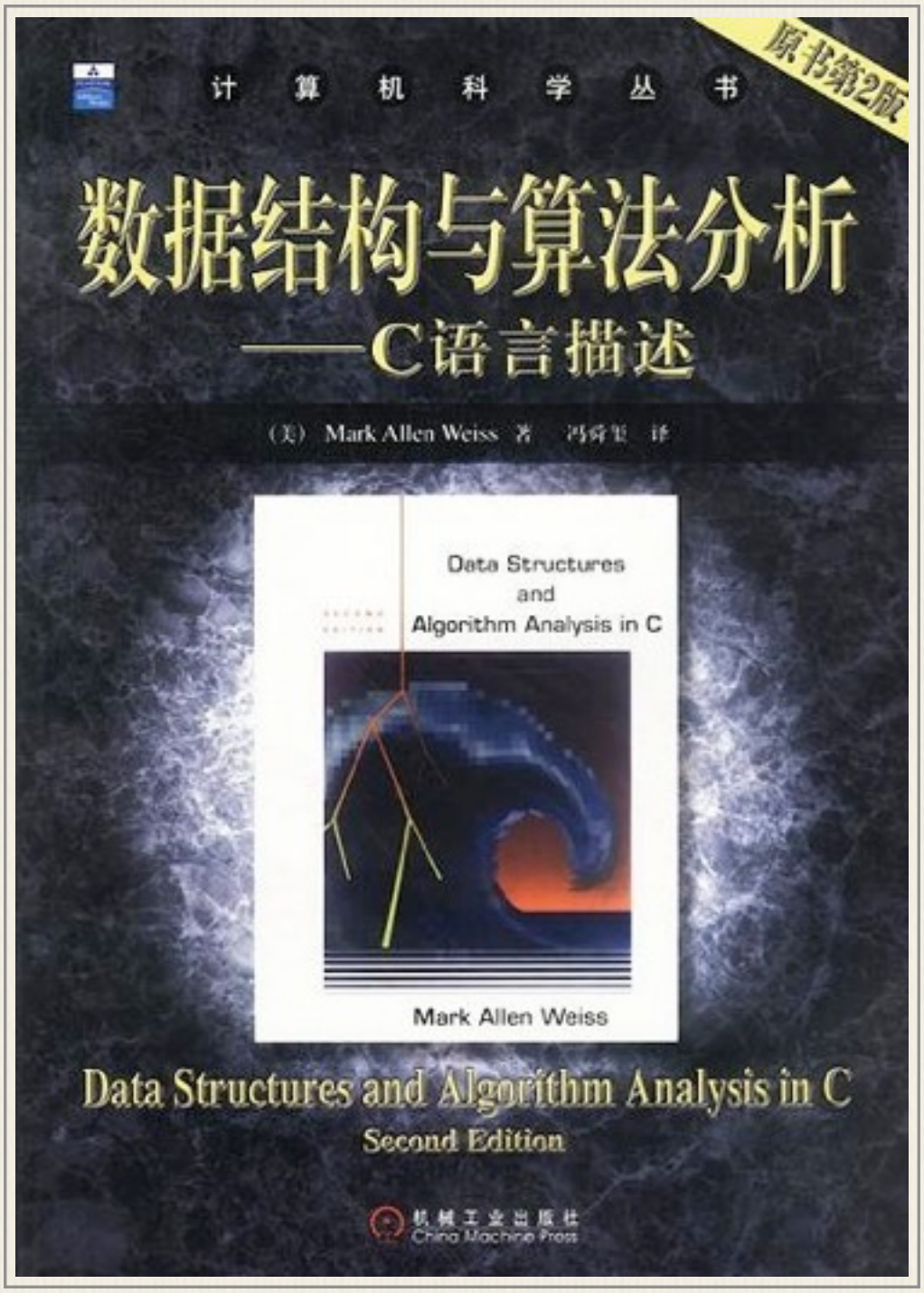
初学

第一次接触数据结构是在大二下学期的数据结构课程。然而这门课程并没有让我入门——当时自己正忙于倒卖各种MP3和耳机，对于这些课程根本就不屑一顾——反正最后考试划个重点也能过，于是这门整个计算机专业本科最重要的课程就被傻逼的我直接忽略过去了。

直到大三我才反应过来以后还要找工作——而且大二的折腾证明了我并没有什么商业才能，以后还是得靠码代码混饭吃，我当时惊恐的发现自己对编程序几乎一无所知，于是我给自己制订了一个类似于建国初期五年计划的读书成长计划，其中包括C语言基础、数据结构以及计算机网络等方面的书籍。

读书计划的第一步是选择书籍，我曾向当时我觉得很牛的”学长”和”大神”请教应该读哪些算法书籍，”学长”们均推荐算法导论，还有几个”大神”推荐计算机程序设计艺术（现在我疑心他们是否翻过这些书），草草的翻了下这两本书发现实在看不懂，但幸运的是我在无意中发现了豆瓣这个神奇的网站，里面有很多质量不错的书评，于是我就把评价很高而且看上去不那么吓人的计算机书籍都买了下来——事实证明豆瓣要比这些”学长”或是”大神”靠谱的多得多。

数据结构与算法分析——C语言描述



数据结构与算法分析——C语言描述是我学习数据结构的第一本书：当时有很多地方看不懂，于是做记号反复看；代码看不明白，于是抄到本子上反复研读；一些算法想不通，就把它所有的中间状态全画出来然后反复推演。

事实证明尽管这种学习方法看起来傻逼而且效率很低，但对于当时同样傻逼的我却效果不错——傻人用傻办法嘛，而且这本书的课后题大多都是经典的面试题目，以至于日后我看到编程之美的第一反应就是这货的题目不全是抄别人的么。

至今记得，这本书为了说明算法是多么重要，在开篇就拿最大子序列和作为例子，一路把复杂度从O(N3)杀到O(N2)再到O(NlgN)最后到O(N)，当时内心真的是景仰之情=如滔滔江水连绵不绝，尼玛为何可以这么屌，

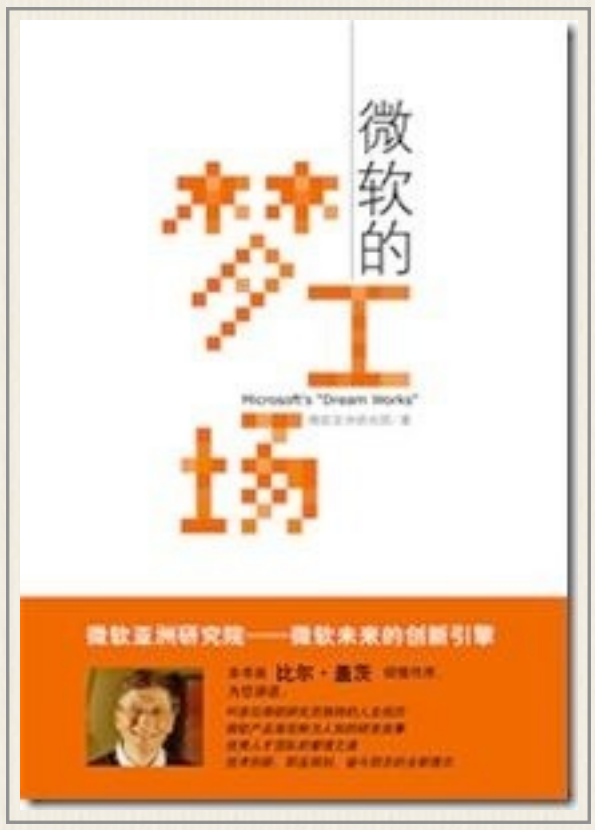
此外，我当时还把这本书里图算法之前的数据结构全手打了一遍，后来找实习还颇为自得的把这件事放到简历里，现在想想真是傻逼无极限。

凭借这个读书成长计划中学到的知识，我总算比较顺利的找到了一份实习工作，这是后话。

入门

我的实习并没有用到什么算法（现在看来就是不停的堆砌已有的API，编写一堆自己都不知道对不对的代码而已），在发现身边的人工作了几年却还在和我做同样的事情之后，我开始越来越不安。尽管当时我对自己没什么规划，但我清楚这绝壁不是我想做的工作。

微软的梦工厂



在这个摇摆不定的时刻，微软的梦工场成了压倒骆驼的最后一支稻草，这本书对微软亚洲研究院的描写让我下定了”找工作就要这样的公司”的决心，然而我又悲观的发现无论是以我当时的能力还是文凭，都无法达到微软亚研院的要求，矛盾之下，我彻底推翻了自己”毕业就工作”的想法，辞掉实习，准备考研。

考研的细节无需赘述，但至今仍清楚的记得自己在复试时惊奇且激动的发现北航宿舍对面就是微软西格玛大厦，那种离理想又进了一步的感觉简直爽到爆。

算法设计与分析

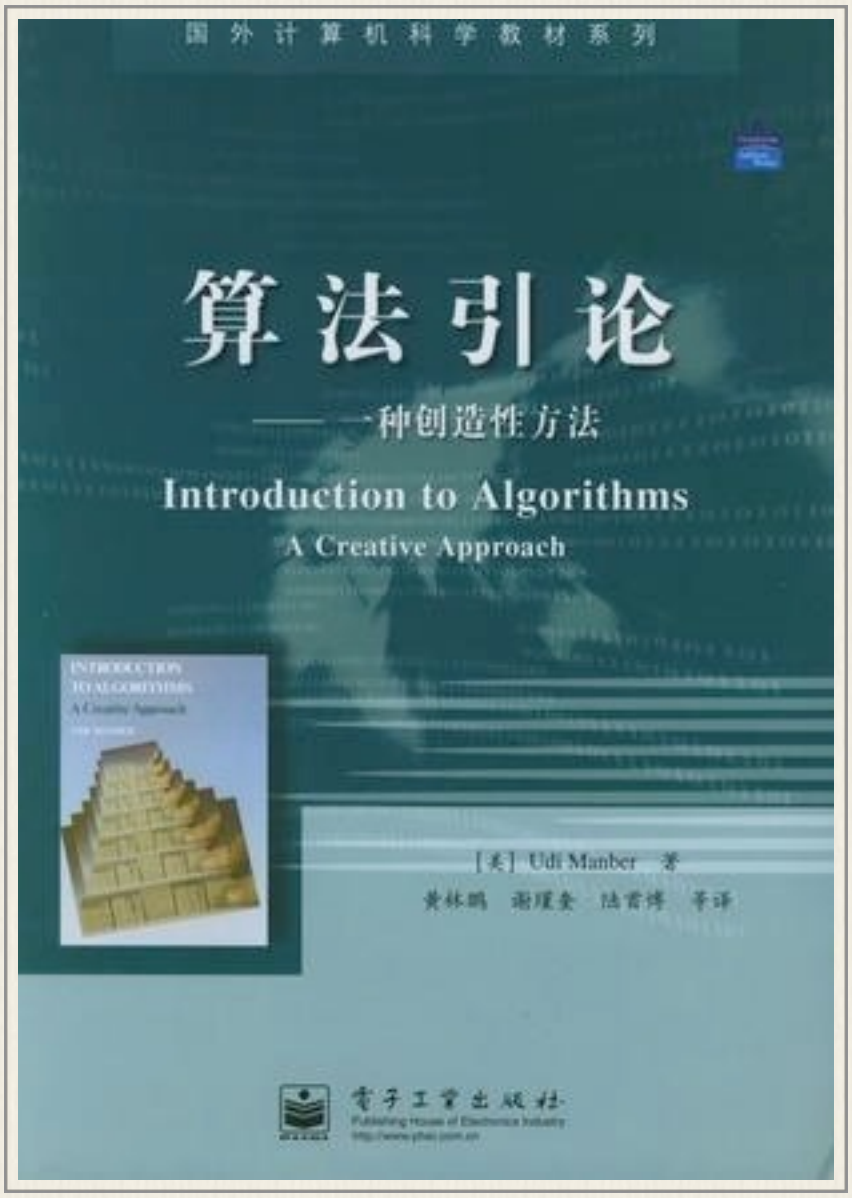
我的研究生生涯绝对是一个反面典型——翘课，实习，写水论文，做水研究，但有一点我颇为自得——从头到尾认真听了韩军教授的算法设计与分析课程。

韩军给我印象最深的有两点：课堂休息时跑到外面和几个学生借火抽烟；讲解算法时的犀利和毫不含糊。

尽管韩军从来没有主动提及，但我敢肯定算法设计与分析基础就是他算法课程事实上的（de-facto）教材，因为他的课程结构几乎和这本书的组织结构一模一样。

如果数据结构与算法分析——C语言描述是我的数据结构启蒙，那么韩军的课程和算法设计与分析基础就是我的算法启蒙，结合课程和书籍，我一一理解并掌握了复杂度分析、分治、减治、变治、动态规划和回溯这些简单但强大的算法工具。

算法引论



算法引论是我这时无意中读到的另一本算法书，和普通的算法书不同，这本书从创造性的角度出发——如果说算法导论讲的是有哪些算法，那么算法引论讲的就是如何创造算法。结合前面的算法设计与分析基础，这本书把我能解决的算法问题数量扩大了一个数量级。

之后，在机缘巧合下，我进入微软亚洲工程院实习，离理想又近了一步，自我感觉无限牛逼。

巩固

在微软工程院的实习是我研究生阶段的一个非常非常非常重要的转折点：

1. 做出了一个还说的过去的小项目。

2. 期间百度实习面试受挫，痛定思痛之下阅读了大量的程序设计书。

3. 微软的实习经历成为了我之后简历上为数不多的亮点之一（本屌一没成绩，二没论文，三没ACM）。

这里就不说1和3了（和本文题目不搭边），重点说说2。

由于当时组内没有特别多的项目，我负责的那一小块又提前搞定了，mentor便很慷慨的扔给我一个Kinect和一部Windows Phone让我研究，研究嘛，自然就没有什么deadline，于是我就很鸡贼的把时间三七开：七分倒腾Windows Phone，三分看书&经典论文。

然而一件事打断了这段安逸的生活——

百度实习面试

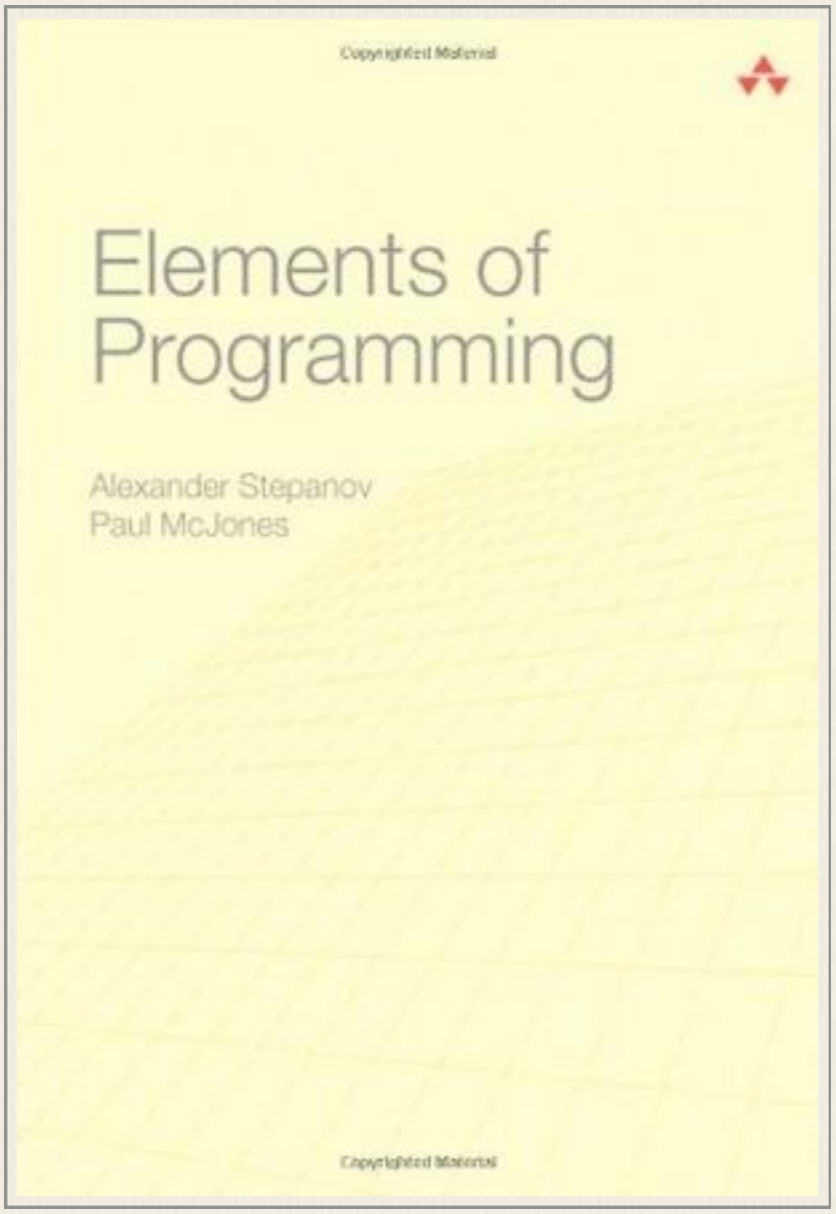
基友在人人发百度实习内推贴，当时自我感觉牛逼闪闪放光芒，于是就抱着看看国内IT环境+虐虐面试官的变态心理投了简历，结果在第一面就自己的师兄爆出翔：他让我写一个stof（字符串转浮点数），我磨磨唧唧半天也没写出完整实现，之后回到宿舍赶快写了一个版本发到师兄的邮箱，结果对方压根没鸟我。

这件事对我产生了很大的震动——

- 原来自己连百度实习面试都过不去。
- 原来自己还是一个编程弱逼。
- 原来自己还是一个算法菜逼。

痛定思痛，我开始了第二个”五年计划”，三七开的时间分配变成了七三开：七分看书，三分WP。而这一阶段的重点从原理（Principle）变成了实现（Implementation）——Talk is cheap, show me the code.

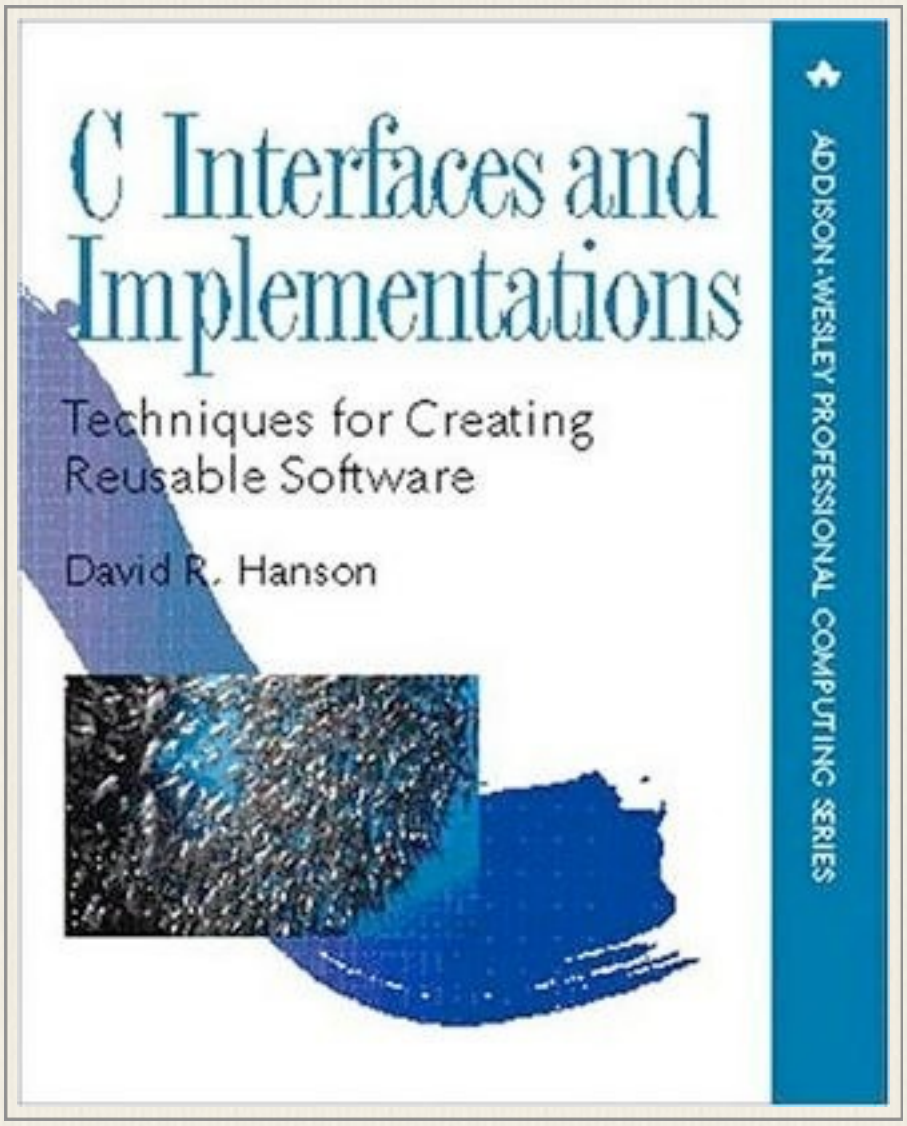
Elements of Programming



由于一直觉得名字里带”Elements of”的都是酷炫叼炸天的书，所以我几乎是毫不犹豫的买了这本Elements of Programming，事实上这本书里的代码（或者说STL的代码）确实是：快，狠，准，古龙高手三要素全齐。

C Interfaces and Implementation

百度面试被爆出翔的经历让我意识到另一个问题，绝大多数公司面试时都需要在纸上写C代码，而我自己却很少用C（多数情况用C#），考虑到自己还没牛逼到能让公司改变面试流程的地步，我需要提升自己编写C代码的能力（哪怕只是为了面试）。一顿Google之后，我锁定了C Interfaces and Implementation——另一本关于如何写出狂炫酷帅叼炸天的C代码的奇书，这里套用下Amazon的评论：Probably the best advanced C book in existance。



严格来说上面两本书都不是传统的算法书，因为它们侧重的都不是算法，而是经典算法的具体实现（Implementation），然而这正是我所需要的：因为算法的原理我能说明白，但要给出优雅正确简练的实现我就傻逼了，哪怕是stof这种简单到爆的”算法”。

依然是以前的傻逼学习方法：反复研读+一遍又一遍的把代码抄写到本子上，艰难的完成了这两本书后，又读了相当数量的编程实践（Programming Practice）书籍，自我感觉编程能力又大幅提升，此外获得新技能——纸上编码。这也成为了我之后找工作面试的三板斧之一。

应用

说老实话，自从本科实习之后，我就一直觉得算法除了面试时能用用，其它基本用不上，甚至还写了一篇当时颇为自得现在读起来极为傻逼的文章

来黑那些动不动就”基础”或”内功”的所谓”大牛”们，这里摘取一段现在看起来很傻逼但当时却觉得是真理的文字：

所以那些动则就扯什么算法啊基础啊内功啊所谓的大牛们，请闭上你的嘴，条条大道通罗马。算法并不是编程的前提条件，数学也不会阻碍一个人成为优秀的程序员。至少在我看来，什么算法基础内功都是唬人的玩意，多编点能用的实用的程序才是王道，当然如果你是一个pure theorist的话就当我什么都没说好了。

然而有意思的是，写了这篇文章没多久，鼓吹算法无用论的我自己做的几个大大小小的项目全部用到了算法——我疑心是上天在有意抽我的脸。

LL(k)

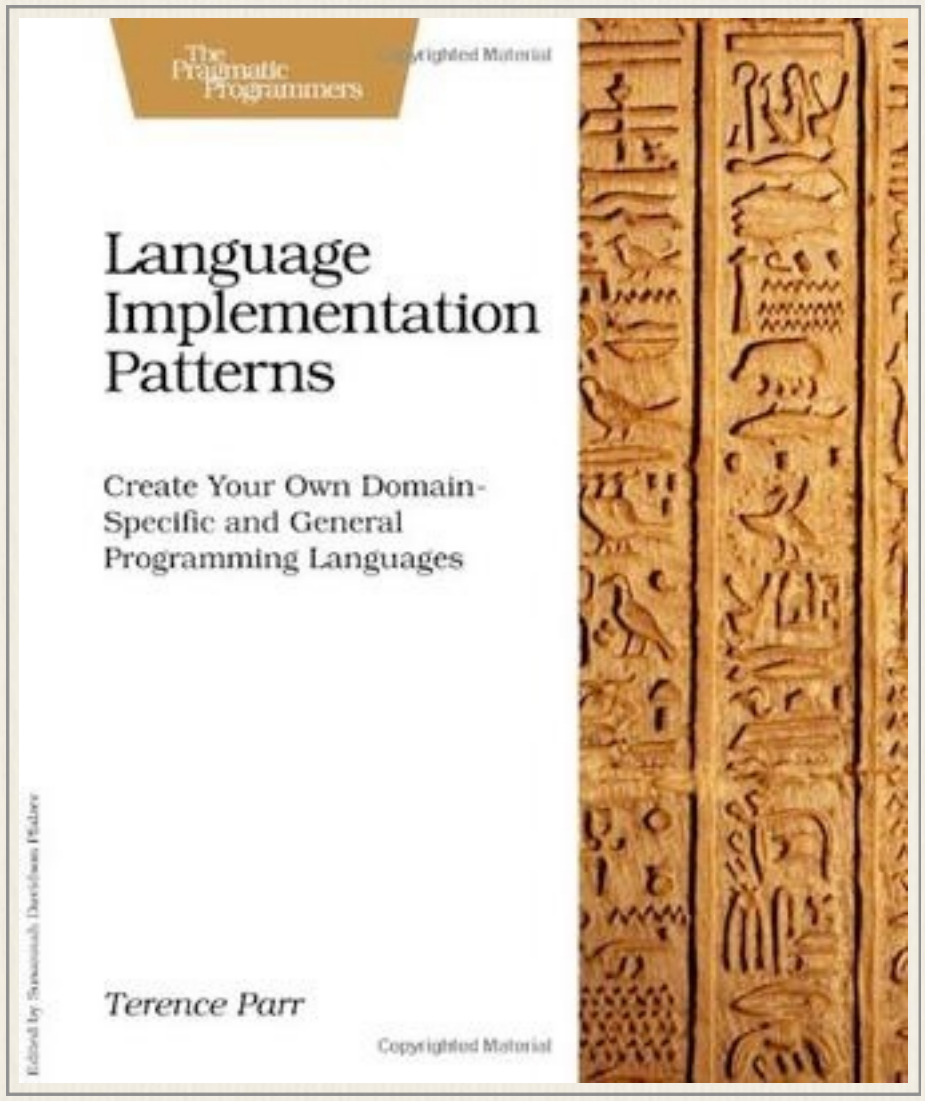
我在微软实习的第一个项目做的是代码覆盖率分析——计算T-SQL存储过程的代码覆盖率。

简单的看了下SQL Server相关的文档，我很快发现SQL Reporting Service可以记录T-SQL的执行语句及行号，于是行覆盖（line coverage）搞定，但老大说行覆盖太naive，我们需要更实际的块覆盖（block coverage）。

阅读了块覆盖的定义后，我发现我需要对T-SQL进行语法分析，在没有找到一个好用的T-SQL Parser的情况下，只能自己动手搞一个：

比较奇诡的是，做这个项目时当时我刚好把ANTLR作者的Language Implementation Patterns看了一半，什么LL(k)啊Packrat啊AST Walker的概念啊正热乎着呢。于是，自己就照着T-SQL 的官方EBNF，三下五除二撸了一个T-SQL存储过程的LL(k) Parser，把代码转换成AST，然后用一个External AST Walker生成代码块覆盖的HTML报表，全部过程一周不到。

老大自然是很满意——我疑心他的原计划是花两三个月来完成这个项目，因为这个项目之后的两个月我都没什么活干，天天悠哉游哉。



拼音索引

拼音索引是我接的一个手机应用私活里的小模块，用户期待在手机文本框可以根据输入给出智能提示：

比如说输入中国：

同样，输入拼音也应给出提示：

中文匹配这个简单，但拼音匹配就得花时间想想了——懒得造轮子的我第一时间找到了微软的拼音库，但接下来我就发现微软这个鸟库在手机上跑不动，研究了下发现WP7对Dictionary的items数量有限制，貌似是7000还是8000个item就会崩盘，而标准汉字则有两万多个，尼玛。

痛骂MS坑爹+汉字坑爹之余，还是得自己撸一个库出来：

1. 首先把那两万个汉字搞了出来，排序，然后弄成一个超长的字符串。

2. 接下来用Int16索引了汉字所有的拼音（貌似500多个）。

3. 再接下来用Int64建立汉字和拼音的关联——汉字有多音字，所以需要把多个拼音pack到一个Int64里，这个简单，位操作就搞定。

4. 最后用二分+位移Unpack，直接做到从汉字到拼音的检索。

5. 后来小测了下性能，速度是MS原来那个库的五十倍有余，而代码量只有336行。

用户很happy ——因为我捎带把他没想到的多音字都搞定了，而且流畅的一逼。

我也很happy，因为没想到自己写的库居然比MS的还要快几十倍，同时小十几倍。

从这个事情之后我变得特别理解那些造轮子的人——你要想想，如果你需要一个飞机轮子但市场上只有自行车轮子而且老板还催着你交工，你能怎么搞。

快速字符串匹配

前面提到在微软实习时老大扔给我一个Windows Phone让我研究下，我当时玩了玩就觉着不太对劲，找联系人太麻烦。

比如说找”张晓明”，WP只支持定位到Z分类下——这意味着我需要在Z分类下的七十多个联系人（姓张的姓赵的姓钟的等等）里面线性寻找，每次我都需要滑动四五秒才能找到这个张姓少年。

这TMD也太傻逼了，本屌三年前的老破NOKIA都支持首字母定位，996->ZXM->张晓明，直接搞定，尼玛一个新时代Windows Phone居然会弱到这个程度。

搜了一下发现没有好用的拨号程序，于是本屌就直接撸了一个支持首字母匹配的拨号程序出来扔到WP论坛里。

结果马上就有各种问题出现——最主要的反映是速度太慢，一些用户甚至反馈按键有时要半秒才有反应。本屌问了下他的通讯录大小：大概3000多人。

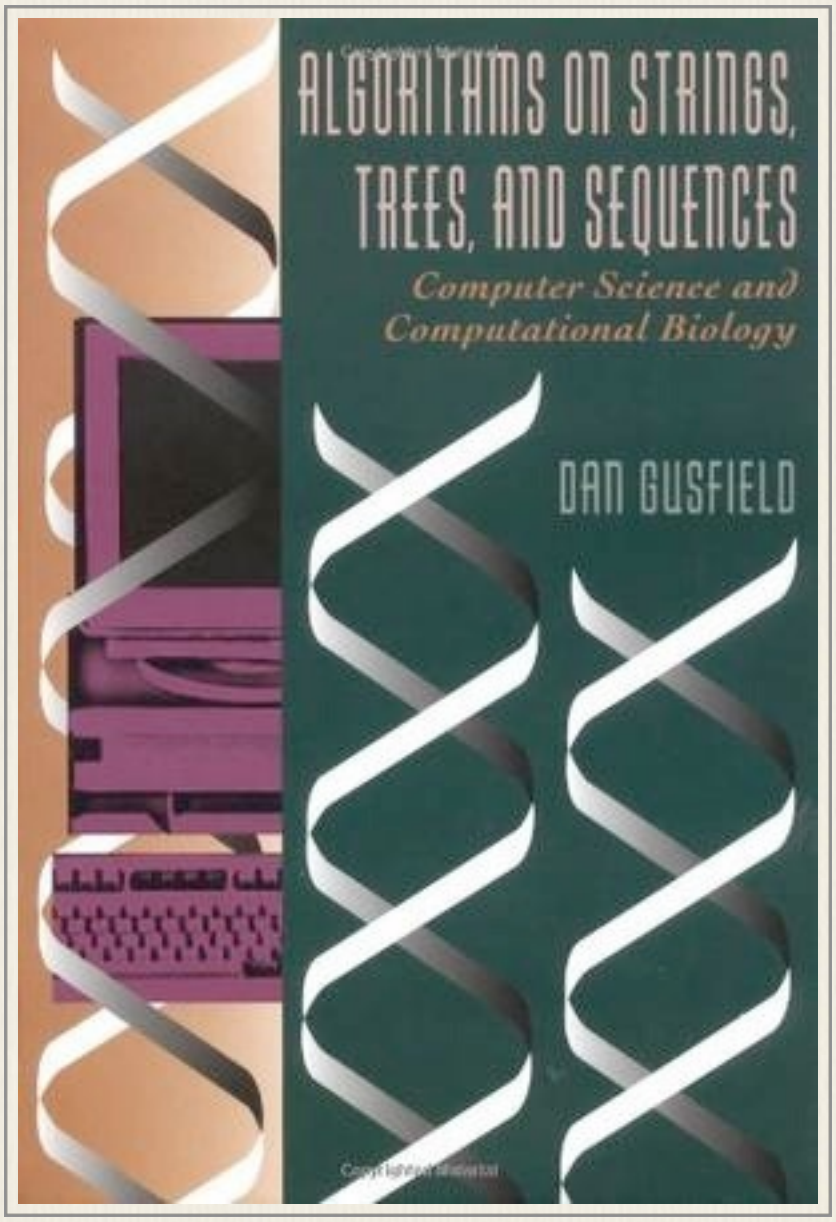
吐槽怎么会有这么奇葩的通讯录之余，我意识到自己的字符串匹配算法存在严重的性能问题：读取所有人的姓名计算出拼音，然后一个个的匹配——结果如果联系人数量太多的话，速度必然拙计。

于是我就开始苦思冥想有没有一个能够同时搜索多个字符串的高端算法，以至于那两天坐地铁都在嘟囔怎么才能把这个应用搞的快一些。

最终还是在Algorithms on Strings, Trees and Sequences里找到了答案——确实有能够同时搜索多个字符串的方法：Tries，而且这本书还用足足一章来讲怎么弄Multiple string comparison，看得我当时高潮迭起，直呼过瘾。

具体细节不多说，总之换了算法之后，匹配速度快了大约九十多倍，而且代码还短了几十行。哪怕是有10000个联系人，也能在0.1秒内搞定，速度瓶颈就这样愉快的被算法搞定。

Algorithms on Strings, Trees and Sequences

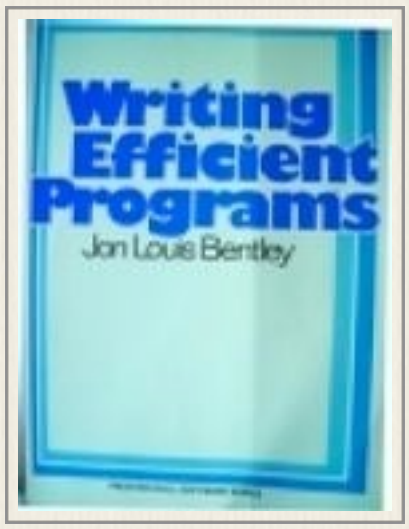


Writing Efficient Programs

之后又做了若干个项目，多多少少都用到了”自制”的算法或数据结构，最奇诡的一次是写一个电子书阅读器里的分页，我照着模拟退火（Simulated Annealing）的原理写了一个快速分页算法，事实上这个算法确实很快——但问题是我都不知道为啥它会这么快。

总之，算法是一种将有限计算资源发挥到极致的武器，当计算资源很富余时算法确实没大用，但一旦到了效率瓶颈算法绝壁是开山第一刀（因为算法不要钱嘛！要不还得换CPU买SSD升级RAM，肉疼啊！！）。一些人会认为这种说法是有问题，因为编写新算法的人力成本有时比增加硬件的成本还要高——但别忘了增加硬件提升效率也是建立在算法是Scalable的基础上——说白了还是得撸算法。

说到优化这里顺带提一下Writing Efficient Programs——很难找到一本讲代码优化的书（我疑心是自从Knuth说了过早优化是万恶之源之后没人敢写，万恶之源嘛，写它干毛），注意这本书讲的是代码优化——在不改变架构、算法以及硬件的前提之下进行的优化。尽管书中的一些诸如变量复用或是循环展开的trick已经过时，但总体仍不失为一本好书。



提高

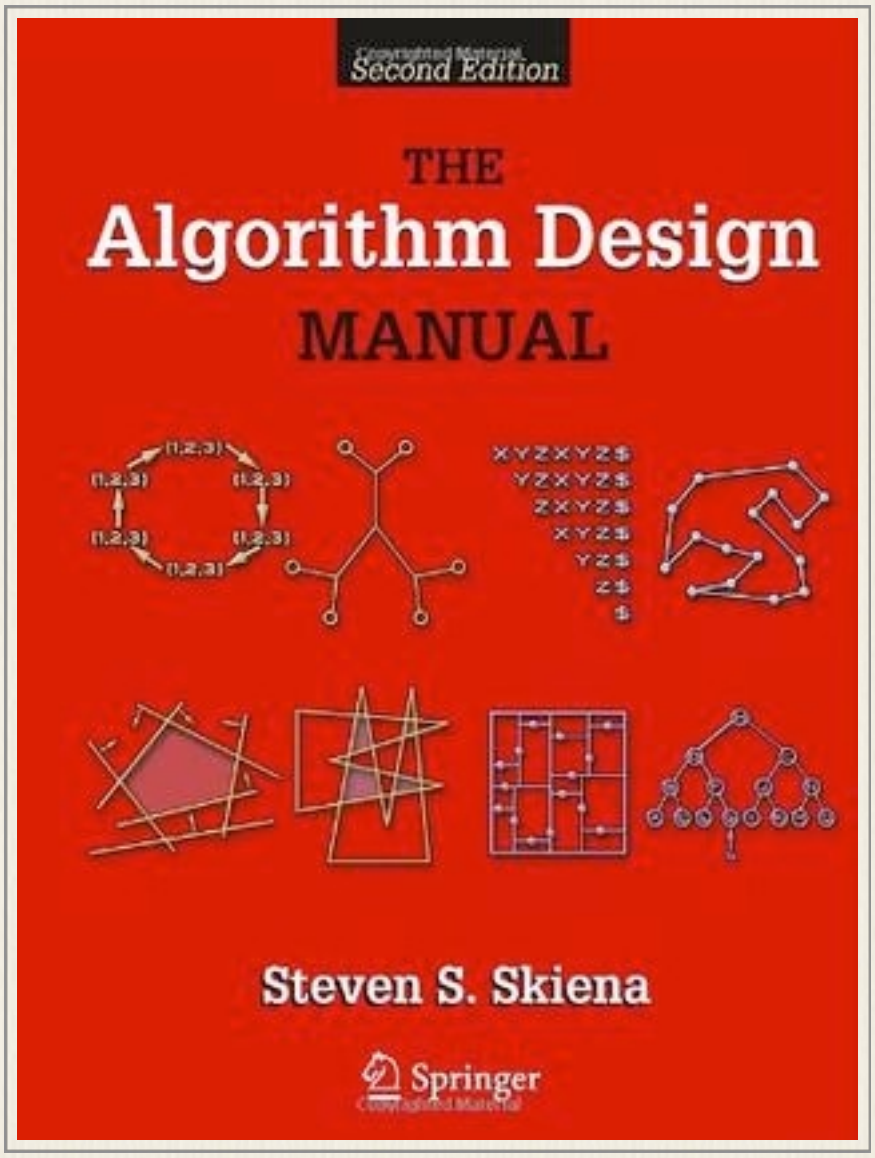
实习实习着就到了研二暑假，接下来就是求职季。

求职季时我有一种莫名的复仇感——尼玛之前百度实习面试老子被你们黑的漫天飞翔，这回求职老子要把你们一个个黑回来，尼玛。

现在回想当时的心理实属傻逼+幼稚，但这种黑暗心理也起了一定的积极作用：我丝毫不敢有任何怠慢，以至于在5月份底我就开始准备求职笔试面试，比身边的同学早了两个月不止。

我没有像身边的同学那般刷题——而是继续看书抄代码学算法，因为我认为那些难得离谱的题面试官也不会问——事实上也是如此。

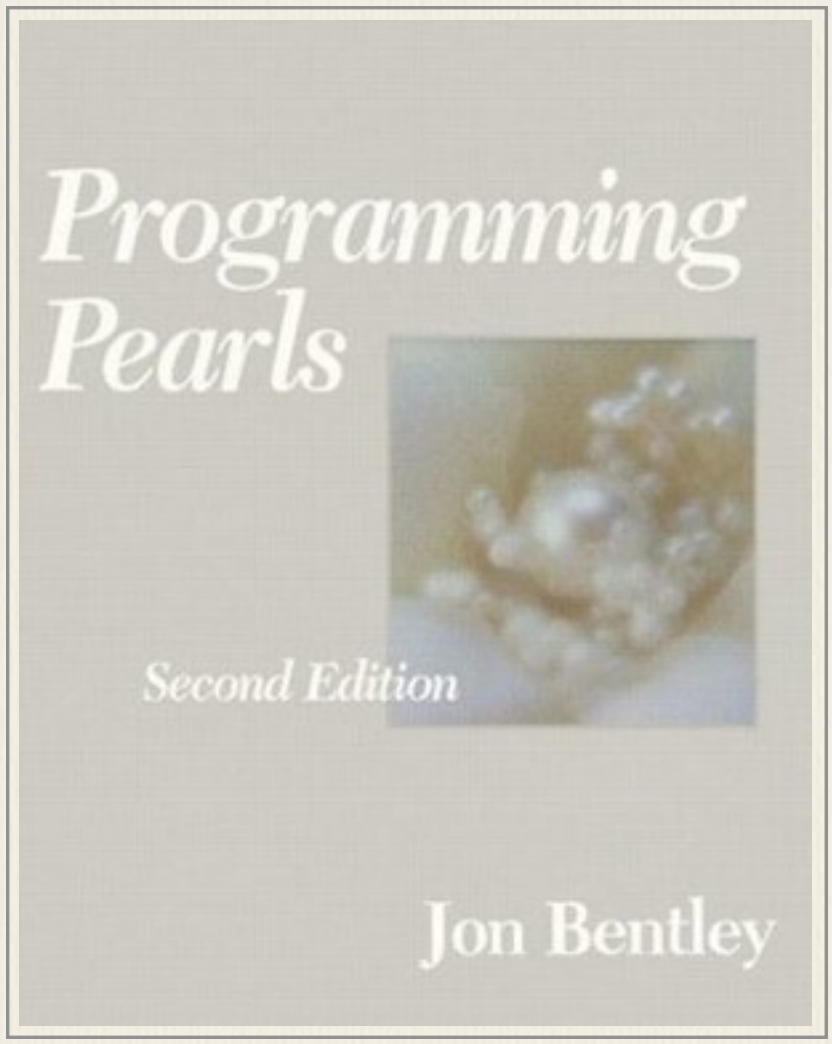
Algorithm Design Manual

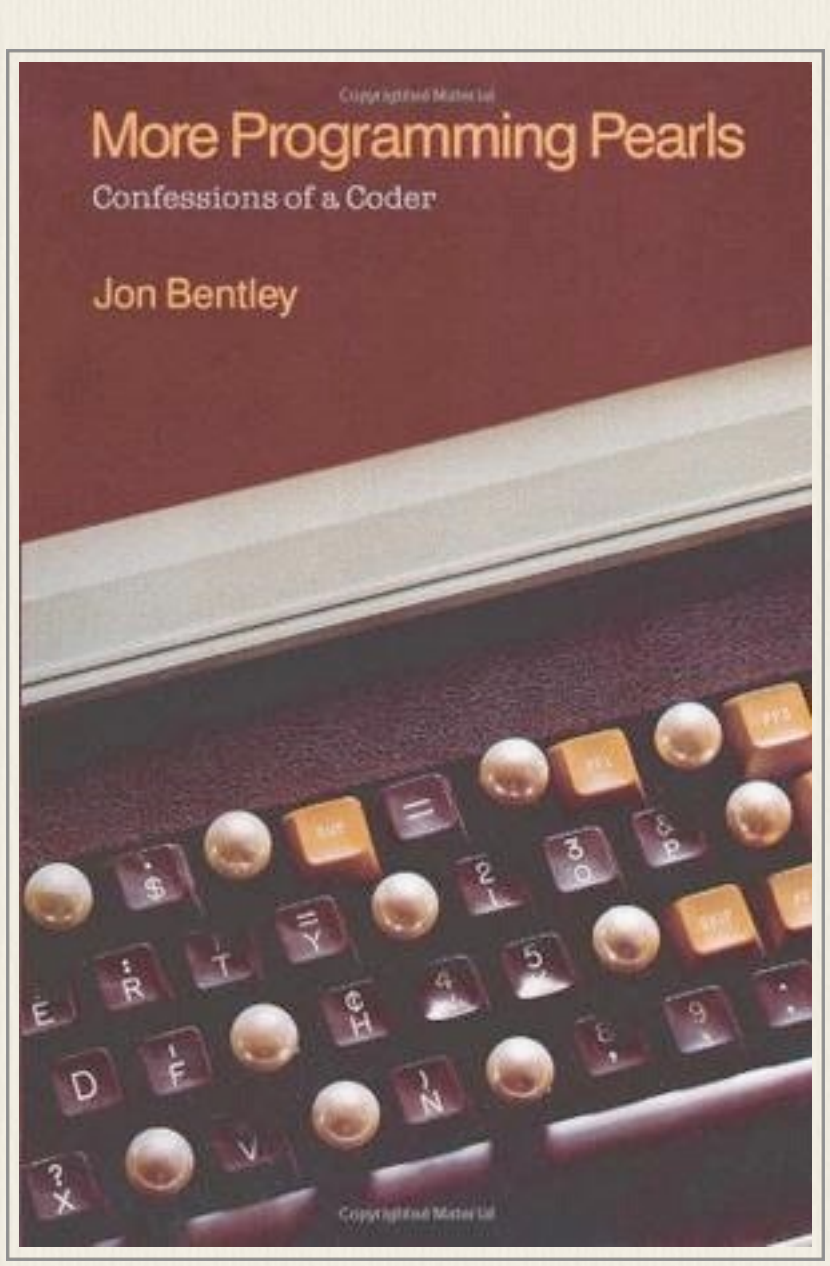


因为很多Coding Interview的论坛都提到这本红皮书，我也跟风搞了一本。事实证明，仅仅是关于Backtrack Template 那部分的描述就足以值回书价，更不用说它的Heuristics和课后题。

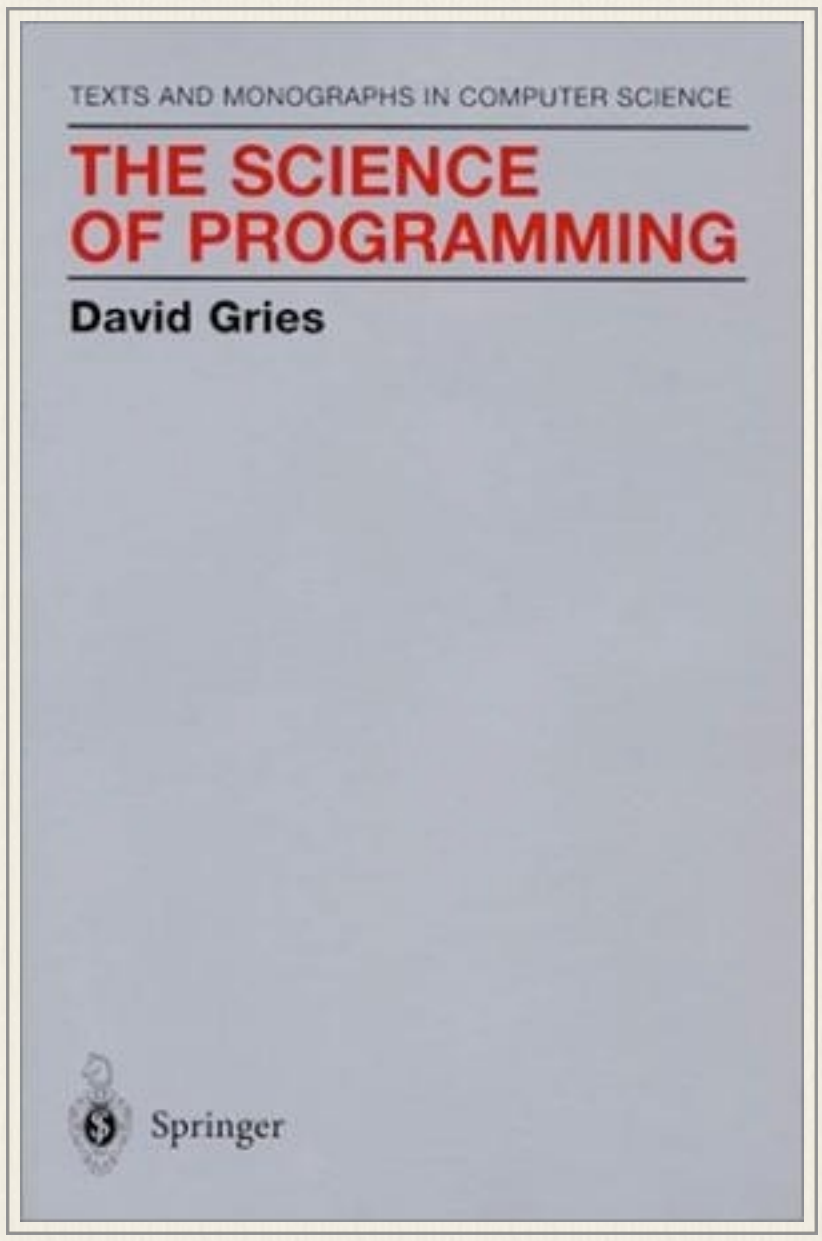
编程珠玑&更多的编程珠玑

这两本书就不用多介绍，编程珠玑和更多的编程珠玑，没听说过这两本书请自行面壁。前者偏算法理论，后者偏算法轶事，前者提升能力，后者增长谈资，都值得一读。


Programming
Pearls
Second Edition
Jon Bentley


More Programming Pearls
Confessions of a Coder
Jon Bentley

The Science of Programming



读到编程珠玑里面关于Binary Search的正确性证明时我大呼过瘾，原来程序的正确性也是可以推导的，然后我就在那一章的引用里发现David Gries的The Science of Programming。看名字就觉得很厉害，直接搞了一本开撸。

不愧为编程珠玑引用的书籍，撸完The Science of Programming之后，本屌获得了证明简单代码段的正确性这个技能——求职面试三板斧之二。

证明简单代码段的正确性是一个很神奇的技能——因为面试时大多数公司都会要求在纸上写一段代码，然后面试官检查这段代码，如果你能够自己证明自己写的代码是正确的，面试官还能挑剔什么呢？

之后就是各种面试，详情见之前的博客，总之就是项目经历、纸上代码加正确性证明这三板斧，摧枯拉朽。

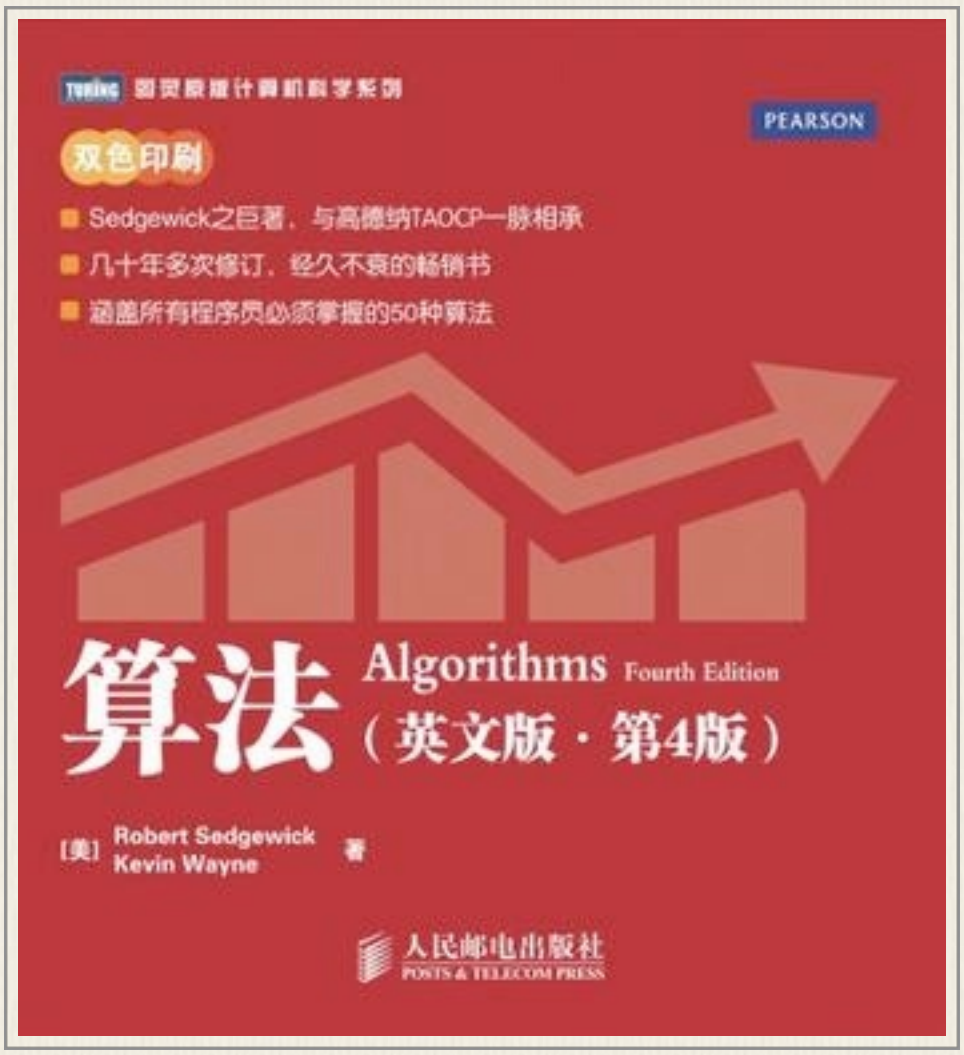
进化

求职毕业季之后就是各种Happy，Happy过后本屌发现即将面临另一个问题：算法能力不足。

因为据说以后的同事大多是ACM选手，而本屌从来没搞过算法竞赛，而且知道的算法和数据结构都极为基础：像那些元胞自动机、斐波那契堆或是线段树这些高端数据结构压根只是能把它们的英文名称拼写出来，连用都没用过，所以心理忐忑的一逼。

为了不至于到时入职被鄙视的太惨烈，加上自己一贯的算法自卑症，本屌强制自己再次学习算法：

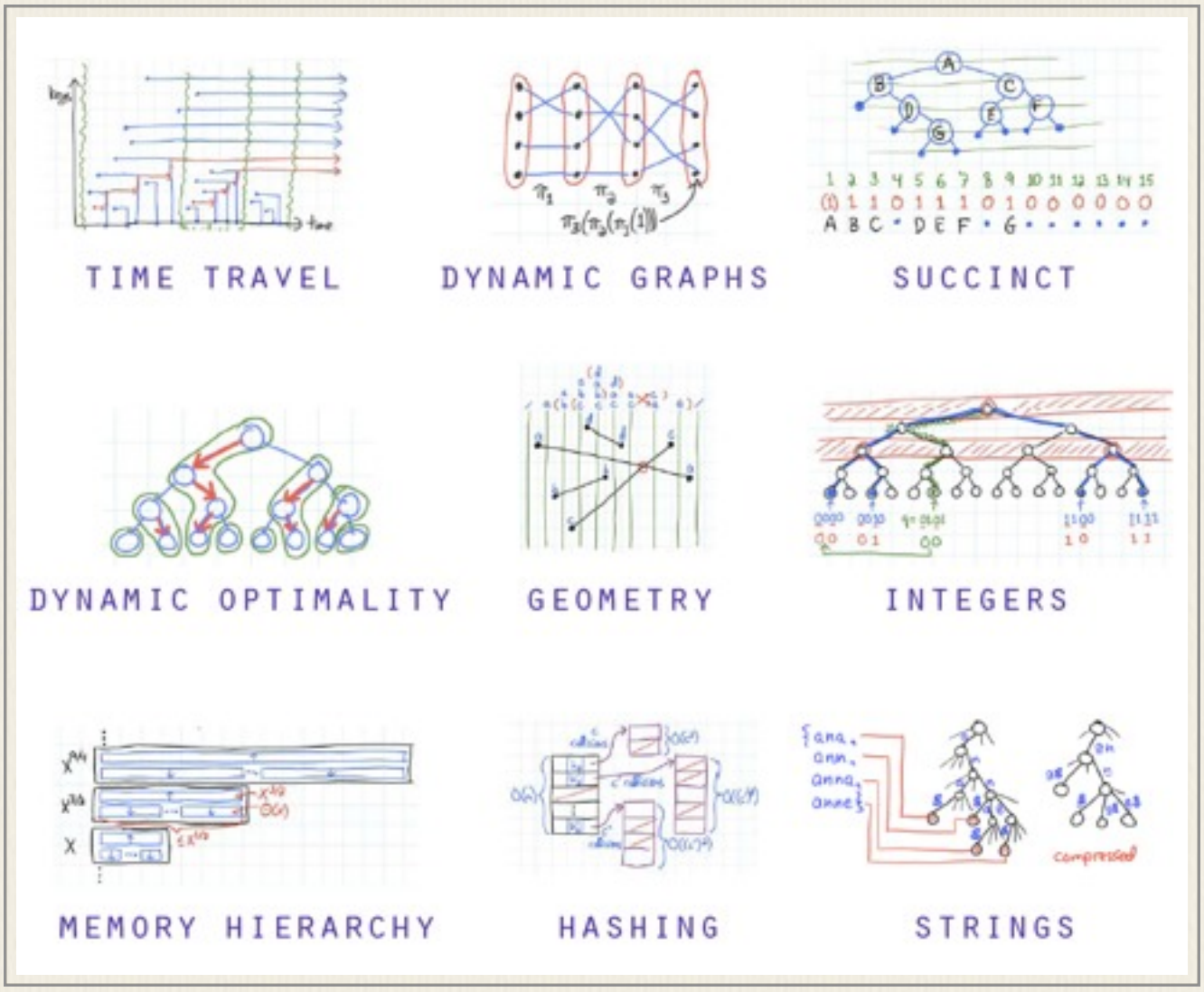
Algorithms 4th



Algorithms是我重温算法的第一本书，尽管它实际就是一本数据结构的入门书，但它确实适合当时已经快把算法忘光的本屌——不为学习，只为重温。

这本书最大的亮点在于它把Visualization和Formatting做到了极致——也许它不是最好的数据结构入门书，但它绝壁是我读过的排版最好的书，阅读体验爽的一逼；当然这本书的内容也不错，尤其是红黑树那一部分，我想不会有什么书会比此书讲的更明白。

6.851 Advanced Data Structures



Advanced Data Structures 是MIT的高级数据结构教程，为什么会找到这个教程呢？因为Google Advanced Data Structures第一个出来的就是这货。

这门课包含各种让本屌世界观崩坏的奇诡数据结构和算法，它们包括但不限于：

- 带”记忆”的数据结构（Data Structure with Persistence）。

- van Emde Boas（逆天的插入，删除，前驱和后继时间复杂度）。
- o(1)时间复杂度的的LCA、RMQ和LA解法。
- 奇幻的o(n)时间复杂度的Suffix Tree构建方法。
- o(lglgn)的BST。
- …

总之高潮迭起，分分高能，唯一的不足就是没有把它们实现一圈。以后本屌一定找时间把它们一个个撸一遍。

总结

从接触算法到现在，大概七年：初学时推崇算法牛逼论，实习后鼓吹算法无用论，读研后再被现实打回算法牛逼论。

怎么这么像辩证法里的肯定到否定再到否定之否定。

现在来看，相当数量的鼓吹算法牛逼论的人其实不懂算法的重要性——如果你连用算法解决实际问题的经历都没有，那你如何可以证明算法很有用？而绝大多数鼓吹算法无用论的人不过是低水平码农的无病呻吟——他们从未碰到过需要用算法解决的难题，自然不知道算法有多重要。

Peter Norvig 曾经写过一篇非常精彩的SICP书评，我认为这里把SICP换成算法依然适用：

To use an analogy, if algorithms were about automobiles, it would be for the person who wants to know how cars work, how they are built, and how one might design fuel-efficient, safe, reliable vehicles for the 21st century. The people who hate algorithms are the ones who just want to know how to drive their car on the highway, just like everyone else.

MIT教授Erik Demaine则更为直接：

If you want to become a good programmer, you can spend 10 years programming, or spend 2 years programming and learning algorithms.

总而言之，如果你想成为一个码农或是熟练工（Code Monkey），你大可以不学算法，因为算法对你确实没有用；但如果你想成为一个优秀的开发者（Developer），扎实的算法必不可少，因为你会不断的掉进一些只能借助算法才能爬出去的坑里。

以上。

原文链接:

http://zh.lucida.me/blog/on-learning-algorithms/

和JING聊聊QUBIT的产品和技术栈

作者:叶玎玎,技术创业者,系统架构师,Fengche.co的创始人,Teahour.fm主播

本文是 Teahour 第 50 期 『和Qubit的工程师聊聊A/B testing, Node 和 Ruby』 的录音文本，欢迎大家订阅 Teahour，iTunes URL 是 http://itunes.apple.com/cn/podcast/teahour.fm/id608387170?l=en。Android 用户可以使用 AntennaPod 来订阅。同时，欢迎加 Teahour 好友，微博和 Twitter。

Part 1 - HackerNews Meetup

叶玎玎：大家好，欢迎收听 Teahour ，我是本期的主持人玎玎。本期由我一个人主持，邀请到了来自英国的 Qubit 公司的工程师董京，来 Teahour 做客。董京，你好。

董京：大家好。

叶玎玎：首先你做一个自我介绍吧，让大家来了解一下你的背景。

董京：我现在在英国时区，大早上爬起来跟叶玎玎聊这些事情还是蛮困难的。因为我平时上班也没有起来这么早。我现在在英国创业公司（Qubit），工作了 3 年多了。之前我还有在 F1 赛车和英国电信工作过，都是技术方面的。

叶玎玎：OK，我跟你认识其实也挺蛮巧的。你在英国生活了很多年，今年回国，在 Twitter 上联系到我，说想组织一个活动——是上海的 Hacker News 的线下聚会。当初你是怎么想到回国时组织这样一个活动呢？

董京：我个人虽然比较了解海外的创业市场，但是对中国的创业的环境几乎完全不了解。我在回国的时候想去了解一下，但是发现没有太多渠道或者机会去找聚会。所以我就突然想到，干脆我就自己从头到尾组织一个，找各个公司去 sponsor，自己一个人去联系 speaker。这其实还是蛮有趣的一个经历。

叶玎玎：对，我感觉这次活动也办得还可以。虽然场地上可能还可以有一些提高，但是总体上来说，你一个人办的也还可以。我记得你一个人联系我后，就自己做了一个页面，也设计得很高大上。

董京：那个页面其实还是蛮搞笑的。如果国内有一个可以组织活动的平台，我希望用那个平台。但是我完全没有找到合适的平台，所以我基本上从头到尾，一个晚上，把后台和前台全部写出来了。

叶玎玎：八卦一下，用什么写的？

董京：后台是用 Node.js 写的。我是这样考虑的，那个服务器很小，我不希望需要用太多的内存，所以我没有选择 Rails 或者是其他大的框架，就是一个很简单的 Node.js 和 Express。

叶玎玎：听起来你是 Node 和 Rails 双修了。

董京：平时我在工作的时候，这两个东西还是用得非常多的。基本上公司的 Rails 构架都是我一个人，后台大约 70 – 80% 是我写出来的，大约有 6 - 7 个 Node 服务器在后面。

叶玎玎：OK，那 Teahour 听众也了解到我们其实已经往 Node.js 社区偏了。开个玩笑。你在英国会经常参加这样的 meetup 吗？

董京：对啊，我经常参加。这边的聚会是非常非常多的。你想了解有什么样的 meetup 可以到在国外还是蛮盛行的网站，叫 meetup.com。在中国还是可以用的，你可以在上面发现很多很多活动，不光光是技术圈的 meetup。还有些关于个人兴趣爱好的活动可以去。可以在周末或者晚上找到很多活动。

叶玎玎：你觉得参加这些 meetup 对你最大的收获是什么？

董京：最大的收获是我认识了很多圈内的人。我发现大家还是有这个需求的，但是没有太多的人去组织。从我个人体验来说，从头到尾办一次这样的活动要花费很多的精力。不单单是联系 speaker，我还要组织场地，要提前预定、要提前到场去检查，这不只是一个周末的事，从开始到结束要花至少两个礼拜的时间。晚上还要操很多心，要保证这个活动的宣传要做到位，后续通知要做到位，有很多很多事情要做到位。

叶玎玎：辛苦。你也提到过，月底会组织一个远程的聚会，可能会更加辛苦。

董京：对。好消息是，我昨天刚和我公司的 CTO 联系过，他说如果公司的 budget 足够，他下个月礼拜五可以跟我确认一下，能不能在六月或者五月回国办一个活动。希望是一个好消息。

叶玎玎：OK。我知道你在组织活动的同时，还在为公司寻找一些 developers，同时还提供一个到英国工作的机会。

董京：是这样，创业的，不管是在美国还是在欧洲，都是非常缺人才的。我们在用各种方法去寻找适合的人才，无论在哪里。像我的同事里，前端和中端的工作组大概有 8 个人，这些人里没有一个是纯英国人。我们中有波兰人、立陶宛人、意大利人，再加上我一个中国人，有很多不同的文化。我们也尝试在美国找工程师，而美国，特别是在硅谷，也是非常非常缺人才。尤其是 startup。有经验的人未必适合做创业的工作。

Part 2 - Qubit 产品和技术栈介绍

叶玎玎：那和我们介绍一下你们公司其实是做什么的?

董京：我们公司做的方面还是挺多的。其中一个目标就是提高电商在线的销售率。这需要通过很多渠道。大家可能对这个市场不是特别了解，在欧洲和美国的电商都希望自己建立一个品牌。在中国大家都要提高在淘宝上的销售率。而在外国都是要在 Google 这个平台，在搜索引擎这个平台上竞争，竞标从而提高网站的浏览量，然后提高销售率。

我们主要通过跟踪用户的行为，做一些个性化的需求的调整，从而提高销售率。我们公司一个产品 —— OpenTag，他是可以管理网站用户跟踪标签(Tag/Script)的产品。我们很早进入到这个产业里。大家可能对这个产业不了解，一般来说，这个主要的产业趋势是在网站上跟踪各个用户的行为。比如风车的网站上面，叶玎玎用到了 Mixpanel 。对我们来说这只不过是一个提供商 (provider)。OpenTag 是一个平台，可以实时更改它们。可以使用 Mixpanel，或者 GA（Google Analytics），或者 KISS-Metrics，即可以实时去更改在线网站跟踪代码，只需要花 5 分钟时间。

对于电商网站来说，这个需求是非常非常高的。在国内虽然有淘宝，但你完全没有机会在其上做个性化的调整。没有办法了解某个用户的具体信

息。而如果你有自己的电商平台，你是完全可以做到的。你可以用第三方的工具，比如说 Mixpanel，或者是 GA，或者 Optimizely，去做一些 A/B testing，去做一些具体的跟踪方案。或者通过一些具体的个性化网站的更改方案。

但问题是，很多中小型或者中大型的电商的开发过程都由第三方开发的，项目周期是非常长的。比如说一个企业公司——乐购，他需要找一个第三方去更改他的网站——添加一个跟踪代码，这个周期起码需要两到三个月。因为他们需要认证、要 approval，然后再等第三方把这个东西做好，还要再测试。

通过我们的 OpenTag，这样一个标签管理的工具，我们可以在 5 分钟内把跟踪代码放到网上，可以做各种的逻辑管理。比如用户一定要从 Google Search 进入网站才能启动这个标签，来跟踪这个用户。在这个产业内，我们做得还是非常大的。我们发布一年半以后，Google 才做了他们的 Google Tag Manager，也是可以管理标签的。大家感兴趣的，可以搜一下。

叶玎玎：你们在找怎样的人呢？

董京：目前，我们非常缺对 JavaScript 了解非常深入的。我们在面试的时候，自己说了解 JavaScript 的人的理解一般还是比较表面的。可能只懂怎么用 jQuery ，写很简单的界面的逻辑。但是我们可能是需要招一些深入了解 JavaScript 的人，比如说了解到 ES6（ECMAScript 6） 版本的一些具体的功能有什么，如 class 和 constructor，而且要了解整个 JS 的趋势。

我们面临的问题是什么呢？我们的很多工具直接部署在客户的网站上，他们的环境是完全未知的，所以我们要保证在各种情况下面我们的代码都可以在客户的网站上运行。这是一件非常非常难的事情。我举一个非常简单的例子，是我们曾遇到一个非常狗血的问题。大家原来应该用到过一个比较老的框架—— Prototype JS。很多电商的网站运用这些陈旧的JS库，会自动 overrides window.JSON 上的方法。而我们经常会处理一些 JSON 数据，而它在 serialize 和 deserialize 的时候处理大的 JSON 非常低效，还会经常出错。特别是 JSON 中有 nested array 时它就完全不能处理。

叶玎玎：对，然后你们是怎么解决这些问题的?

董京：我们有自己的一个测试平台，在内部里面可以每天晚上实时运行一些 JavaScript 测试代码，调用各种不同的开发环境在客户的网站上运行，通过测试来达到部署前的测试和部署之后的监控测试，遇到问题之后具体个别处理。

叶玎玎：OK，听起来就是说你们对 JS 的要求还是比较高的。是比较纯粹的 JS 开发。

董京：对，是一个 full cycle，因为我们通过跟踪用户，是通过 JS 来的。然后再回到后台，整理用户数据。但我们另一个目标是如何通过行动来改变在线销售率。很多做分析的网站，比如 GA，或者 Mixpanel，他们的目标倾向于分析。但在很多情况下，分析完之后，如果要做行动、要怎么样改进一个网站还是一个未知数。我们的目标是提供一个简单的解决方案来帮用户提高销售率。我们偏向于行动，不仅有数据分析，而且还有工具可以评价如何让你尽快的做实实在在的更改。

叶玎玎：这一点我还是挺有兴趣了解的。如你提的问题，GA 也好， Mixpanel 也好，他只是告诉你了一个数据，比如这个页面到下一个页面的 flow 转换了多少，用户访问了哪些东西。而我确实知道了一些问题，但是如何解决这些问题。我只能去猜，只能去试，之后看转化率有没有提高。你说用了一些解决方案，能大概举一些例子吗?

董京：我们大部分的客户都是电商，所以很多解决方案都是跟电商相关的东西。比如说一些客户的网站上，用一些 personalized 的方案。是在 homepage 上面提醒明天会下雨，以此接近用户。帮助产品重新做一个包装，让他们可以比较贴近用户一些。这是其中一种方案。另外一种方案，是在一些产品上面显示是不是已经快没有存货了。比如只有两到三个存货的时

候，我们可以做一个界面上的更改，提醒用户，这个产品有很多人买，已经只剩下两个货物了，你是不是要买它。经过测试，根据用户的个性和需求，再加上产品的销售量提示会增加很多销售量。比如一个产品的销量提高2%。

对于大多数的电商来说，2% 的提高是非常大的。我们的客户有 Arcadia Group，其中一个出名品牌叫做 Top-shop，是专门给女孩子卖衣服的。如果提高百分之二的销售率对它来说是非常有利的。

叶玎玎：了解。你们做的是电商垂直，所以你们说有关于电商的很多数据。通过这个数据提供一些建议。

董京：对。我们一个未公开但是已经部署的产品也倾向于 enter-prise。如果要做比较的话，我拿 Strikingly来作例子。大家应该对 Strikingly 比较熟悉，是蛮有趣的，由中国人开发的产品，但是被 YC 投资了。

叶玎玎：是我们已前的嘉宾。『28期：和Strikingly的CTO Dafeng聊聊他们和Y-Combinator的故事』

董京：对。以我个人的见解来说，他们的产品与外国的一个产品叫做 Webflow，是非常非常像的。而且我觉得，Webflow 稍微做的比 Strikingly 专业一些，有很多页面调整的功能是可以更改的，UI 界面也做的非常好。我不知道他们解决的是一个什么问题。对于我来说，Webflow 和 Strikingly 是二十一世纪的 FrontPage。他们把 FrontPage 移到了网络上，但是没有解决用户的需求。如果我是开发人员，我绝对不会去用 Webflow 或 Strik-ingly 去做网站。我有能力去用 offline 的工具开发，而且我觉得更加顺手，所以我不需要用他们的工具。如果我是一个 designer 的话，我也不太会用这个平台，我会用 PS，或者是其他的 offline 的工具。对一个不懂设计的人用这个网站，你也不能完全保证，他能把这个网站做到怎么好。就是说，这

两个产品都是在一个中间层，面对的用户都不是特别的确定。所以我对这种产品稍微有一点怀疑。

叶玎玎：这里说一下我自己的见解。他们的用户群是很确定的。开发设计网页需要一个比较好的 layout，让有美感的，但是不会做网页的人，去做自己的设计。他适合用于展示页面，比如说演员或者模特，他知道怎样的东西是好看的，但是他不会做网页，他可以通过拖拽加一些东西做一些很好看的网页。包括一些摄影的、一些学生。学生可能要给自己做很简单的网站，包括一些页面，来做产品的展示。他有比较确定的用户群体。这类用户是想做网站，而没有做网站的开发能力，所以要找一个相对来说比较简单的工具。

董京：所以是他本身有一些设计的功底，或者一些设计的天分，然后只要把这个页面做好，再加上内容展示。在商业角度上来说，我可能觉得这一部分虽然还是有机会挣钱的，但不是特别多。毕竟以我的背景来说，我更倾向于 enterprise ，这些小的商业模式赚得钱可能会少一些。

叶玎玎：OK，这个可能就不太确定了。我觉得应该发展得很好。

董京：其实我还是蛮喜欢的。我们公司有个产品还是类似于他们的。我们对整个网络有另外一种看法，所有的网站设计趋势都是忘组件化发展的。大部分网站都是可以按模块来算的。不管是图像、文字、段落、复杂的 animation，或者是 slide，都是按模块来算的，你不能跳出模块。我们是以这个角度看待网站开发的。

我们的一个解决方案叫 Deliver，通过分析更改网站的任何一个 component。对于一个开发者，专业地开发一个方案，放到市场上。然后市场的人通过调用开发人员已经开发过的方案部署到网站上面。这样就完全做到专业的人可以用专业的线下工具开发，再用我们的接口模块和工具，指出这个组件在哪个网站可以运作，能够提高多少销售率。对于那些非专业，没

有太多 design、设计方面天赋的人就可以通过 drag-drop 来测试一下这个组件是不是对电商的品牌有利。

这个产业——做 A/B testing 还是有很多公司的。另外一个做得比较好的叫做 Optimizely。我不知道叶玎玎有没用过这工具。

叶玎玎：我觉得好贵。（作者注: 记错产品价格了，Optimizely 的价格还是比较合适的）

董京：如果想用免费的工具，Google 还是有一个这样的工具的，也有 A/B testing 的功能。

叶玎玎：说起 A/B testing，你们是怎么做的呢?

董京：我们的结构非常复杂。你想了解前端还是后端? 做 A/B testing，主要靠 JavaScript，实时地替换页面。有 50% 的人可以看到 B 的页面，有 50% 的人可以看到 A 的页面。

叶玎玎：我比较贪心一点，想了解前后端。我想去尝试一下，但是没有实战的经验，时间上也不太允许。但我想学习这方面的知识。你们是做分析的，还给用户提供建议，对于怎样更好地提高转化率，你们自己也会很多的 A/B testing 吧。

董京：恩，是的。我们内部也做了蛮多的 A/B testing。我们会用自己的工具去做。但还要通过从非专业和非技术的角度上去改善产品。比如我们的产品经理会假设，如果这个页面的模块更改成什么样子，也许会提高百分之多少的销售率，或者达到目标。这个目标需要自己定义。对电商来说，我的目标就是用户买完东西，到最后的 checkout page，然后到 payment，最后 confirmation。但是对于其他用户来说，可能希望用户到注册页面、希望更

多的人填自己的邮件地址，subscribe 我的信件。也可能是，让大家点击一个链接，或者是滚动页面到最下面的位置。可以有很多不同的目标，所以要有不同的途径去实现目标。

叶玎玎：那一般来说在前端，根据目标可能会有不同的板式，或者不同的操作过程。那你们是如何保证样本的独立性?

董京：大部分跟踪是通过 cookie 的，一般都是用 first party cookie。在欧洲，用第三方的 cookie 有很多限制，不能做太多的跨域名跟踪。一般都是在同一个域名下做跟踪。当这个用户第一次登上这个网站之后，会有一个自己的 id。其实可以在淘宝上看到，登陆后用户会有一个 cookie ip。再具体点，用户每一个动作会发到我们的服务器上，我们有一系列的后台运算。我们有专门的做数学模型的博士在我们公司做运算模型。通过运算模型，再把数据发到后台。我们用 Hive、Hadoop、Storm ，来做一些实时的分析。这还是一件非常困难的事情。

叶玎玎：再请教一下，你们后台的 A/B testing 是对前台做一个限制还是怎样?

董京：用 cookie 去实现 A/B testing，其实还是很 limit 的一种方式。大部分企业用户需要了解整个用户的全部的浏览历史。比如说我们有很多用户的数据，来预测一个用户一生会在网站上消费多少。通过这些分析，才能做一些具体的 A/B testing 。比如说我要测试一些用户，这些用户的一生会在这个网站上花超过 1000 英镑，针对这些用户，我们再做一些具体的 A/B testing。给他一些优惠，给他一些 promotion，让这些用户群体上受到一些特别的待遇来测试。我们需要通过跟踪，通过一些后台的数据分析用户的历史。

叶玎玎：相当于不是在代码级别的分片，而是对于一些过去行为在后台计算得到结果，推算他会不会做出一个购买行为。

董京：对我们公司来说，前台后台还是区别蛮大的。如果说前台工作的话，不光光包括前端代码，还包括 middle stack ，全部都要做。对于我自己来说，我平时的工作都是从写 Puppet 开始。如果大家不是很了解的话，就是管理服务器的脚本。从管理服务器开始，到 application logic 再写到前端，是一个非常辛苦的工作。

叶玎玎：全栈。

董京：恩。要切换自己的一个 context 还是非常困难的。

叶玎玎：你们现在开发有多少人?

董京：我刚加入的时候整个公司就一层楼，大概有 28 个人。现在公司已经有两层楼了，整个公司从美国到英国到法国还有德国加起来一共有将近 90 个人，工程师可能占一半的数量。前端工程师有将近 10 个，包括 manager。其他人都是做后台的。后台这部分，我们在欧洲还是做得非常大的。按数量来算，我们每天要收到 1.6 个亿的数据点。

叶玎玎：这么大的数据是怎么处理的。介绍一下你们存储这方面?

董京：我不能介绍太多，因为我不是主要做这个方面的。我可以从层面上说一下。如果你了解 Mixpanel 的数据模型，他们没有数据结构的，都是通过 event 来做的，会比较简单 (scalability的原因)。这个行业里大家都要跟踪用户行为，但是没有一个公司去定义一个数据结构。之前我参加了一个项目，是由我们公司首先提出的一个 standard，让大家都是要这个数据结

构。运用这个电商数据结构提高跟踪的效率、跟踪代码的更改效率，这样你可以从不同的平台上做切换，也可以更好的做一些网站应用的 implementation。

我们这个 specification 叫做 Universal Variable，这个通用变量里有描述到用户的行为，页面的产品，页面的 type/category，比如是 homepage、产品的页面、checkout、basket（购物篮）。这个 spec 已经被列入到 W3C，你可以在 GitHub 上看到 specficaiton。有很多公司都是参加到了这个 spec 的审评的过程，有W3C、Google、IBM。当然还包括我们的 Qubit，也是主要的项目 leader。

叶玎玎：那你能简单介绍 Qubit 的后台用到了什么？你刚才说每天有 Billion 级的数据，如何 scale?

董京：我们有很多 data center，大部分都是在 Amazon 上面。最早是前台的 JS 发数据到后台，接收这个数据。接收端最早是用 Node 写的。当时我们大概可以 handle 的数量在几百万这样。当时 Node 版本是 0.4。在去年，我们做了一个实验，把它换到 JBoss 上，才 handle 到 billion 上。

叶玎玎： handle billion 是指每天的接收的 data point?

董京：对。还是不需要太多的 backup server 才能做到。到后面我们发现 Node 很难 debug，经常有一些 memory 的问题。特别是在早期 0.4、0.6 的时候。虽然接收数据、forwarding 很简单，但是做了实验后还是放到了 Java 平台上。

叶玎玎：还是 Java 更靠谱一点。

董京：对。

叶玎玎：你们有尝试最新版的 Node 吗？测过最大的时候能达到什么程度？

董京：没有尝试过，因为大家都很忙。

叶玎玎：已经受伤了是吧。

董京：对。然后从接收数据到这个模块，再存到 HDFS 上面。之后我们有一个程序把这些数据实时发布到 Hbase、Storm 和 Hive。再做一些运算，根据具体的数据结构的需求把它写入到 Hive 里面，再通过 Storm 和 Hbase 做一些量的分析。

叶玎玎：OK，听起来主要是 Hbase 做数据库存储，Hive 做类 SQL 查询这样的东西。

董京：其实中间还有很多的技术。比如说 Cassandra，做一些 queue 的处理。我们原先用 Cassandra 遇到了一些蛮奇怪的问题，说实在没有办法用下去了。

叶玎玎：能吐槽一下么？

董京：这个我就不能吐槽了，也是听别人说的。这么大的数据玩起来还是挺有意思的。有很多 challege。因为这些数据还是有很多 noise 的，所以要做很多测试、删除，包括有一些 bots，垃圾信息，全部都要隔离开。

叶玎玎：那有个问题我想了解，你们拿到数据后处理，到最后得到的结果有多少延时?

董京：大概做到 5 - 10 分钟。

叶玎玎：那计算还是相当快的。

董京：相当快。我们在 Amazon 上面就有将近 300 台服务器。而且不包括 Spot Instance。

叶玎玎：这个就不是一般人能玩的了。你之前聊到 Node 和 Ruby，特别是你是团队里面唯一一个用 Ruby 的，而其他人都在用 Node?

董京：可能这样说有点太过了，可能公司里 10 个人有 2 - 3 个会 Ruby。但是根据实际情况，我们的工作主要和 JavaScript 有关，所以 JS 的开发人员相对多一些。总体倾向于 Node.js。一般同事只懂一个语言，就是 JavaScript 。也不是说这样不好，大家都用自己知道的东西做事。对实现功能和 startup ，也没有太多的不好。用自己不习惯的语言做一些实现功能，毕竟不会做得特别好。

叶玎玎：那你们有这么多人，为什么还坚持换到 Java?

董京：因为写数据接收的那一块是属于后台的工作，而不是属于前台的工作。我们前台主要是做一些产品方面。后台存储处理和前台关系不大。

Part 3 - Ruby、Node 的比较和欧洲创业社区

叶玎玎：难得碰到一个 Node 和 Ruby 双修的。我们在以前（45期：和《深入浅出Node.js》作者朴灵一起聊聊Node.js）采访朴灵的时候，由我和 Terry 两人主持。而我们对 Node 都不太了解，只好让朴灵来说。你两边都玩，那么你对这两个语言怎么看?

董京：我个人还是比较偏向 Ruby 的，希望 Ruby 稍微给一点力。但是以现在的趋势来说，Node 的还是很有希望的。比如 ES6 的 sppec，发展趋势蛮好。但是我对这个社区不是特别感兴趣。虽然有层出不穷的 library，但是真正写得好的还是很少，特别是做过单元测试的。但是有一些好东西，比如 ES6 添加了很多编程语言的功能，比如 constructor，可以写 OOP。Node 可以做 kernel work、annotation 和 dependency injection，慢慢成熟。我前两天还看到有些人在 ES6 上写了一些 spec——怎么样让 Node 实现多线程，实现平行计算。还是很有希望的社群。

叶玎玎：有没有感觉，在欧洲 Node 社区在不断发展，Ruby 社区有点相对 go down?

董京：有这个趋势。但是我在跟很多创业公司谈的时候，大部分创业的人还是在用 Ruby。有个比较大的创业公司叫 state.com，他的投资方是因特网之父(Tim Berners-Lee)。这个公司在英国，我之前也和他们开发人员聊过。他们用 Ruby，也用 Node.js。Node.js 不做太多后台的数据处理，主要倾向于前台的渲染。state 做的不是 single page project，需要考虑 SEO。但是通过 Node 做一些优化。大部分的后台的 API 都还是用 Rails 写的。

叶玎玎：这个挺有意思的。Node 在中间，然后在前端后端，相对于在中间做了一个桥。

董京：其实我们公司差不多也是这样类似的构架，我们虽然没有利用 Node 做 SEO，但是我之前说过大概有 6 - 7 个程序，6 - 7 个 logic 都是在用 Node 写的。比如一个 proxy 是用 Node 写出来的。但是也是非常困难，我们需要为此做一些 process management。Node 是一个 single event loop，要保证程序 crash 以后不会影响到其他的用户，需要做很多高端的优化。

叶玎玎：了解。这个是 single event loop 的缺点。说到英国的创业环境，我想要了解一下。我可能知道的不多，但是像我们风车在用的 Pusher，支付上用的 Stripe，都是英国的创业公司。所以英国的创业公司是怎样的?

董京：英国的创业环境应该是很好的，大家的趋势都是互帮互助。我发现一个非常好的现象，大部分有创业精神的人，都不是从天而降的一个点子，而是从自己的需求开始。有一个比较出名的打出租车的公司叫 Hailo，在英国是一个做的比较大的公司。它是用手机打车的一个工具。我之前和他们聊的时候，他们的需求也是从自身的需求开始的。这个创业者，家里两代人都是开出租车的。一开始宣传的时候都是从自己家人开始宣传的。比如说我爸爸，我妈妈他们都需要提高载客的时间，从这个地方开始宣传的。

还有一些比较好玩的，是在欧洲的一个建立旅游行程的工具，叫 CityMapper。用来标记从这个地点怎么坐公交车、坐地铁到另外一个地点。这个需求也是从自身开始。原来没有什么工具能做到很准确的推送和预测，他就从头到尾写了这样一个工具。到后来，这样一个工具，在法国和纽约都比较热门。我身边的朋友都在用这个工具。

叶玎玎：那英国的创业环境和欧盟的其他国家比起来怎么样？我弄欧盟签证的时候，除了英国不行，其他都行。

董京：去欧盟其他国家其实还是蛮方便的。你可以申请法国的签证。法国属于欧盟，会给你一个比较长时间的签证，然后用欧盟的多入境的签证，1 年的时间内可以在欧盟里随便进出。

叶玎玎：但是，除了英国。

董京：对，除了英国。

叶玎玎：这很头疼。那对于整个欧洲来说，英国的整个创业环境处在哪一个层次?

董京：我不敢说其他城市，伦敦的环境还是非常好的。在伦敦有很多很棒的创业公司。非常非常多，我都没有办法数了。我原来参加活动的时候，和一个人在台下聊过，1 年后他已经在台上聊自己的产品了。所以我真的很佩服外国人。他们实在是太有魄力了。

叶玎玎：既然你们考虑在国内招人，然后让他人肉翻墙。那么介绍一下英国的衣食住行。

董京：衣食住行的话，一般来说，做工程师的话，工资都是在 30000 英镑/年。一般的初级工程师是这个价格。最贵的东西是房租和交通，吃饭上花的钱不是特别多。我觉得在上海吃饭的价格跟英国差不多了。但是伦敦的房租很贵，地铁也很贵。大概的价格是这样的。我现在一室一厅的房子的房租大概是 1000 朝上一个月。英国地铁是分区的，从 1 区到 6 区。我买了 1 区到 2 区的地铁票，包月是 120 英镑。地铁票每年都在涨价，去年是 116/月，今年是 120/ 月。所以说这都是大头。吃饭的话一个月如果你比较省的话，大概 500 英镑以下就可以做到。

叶玎玎：按照你这个介绍的话，junior 的工资是 30000 英镑/年吧。

董京：我找工作的时候对银行和金融没有特别感兴趣。如果做金融的话，junior 的工资相对高一些，大概是 40000 英镑朝上。

叶玎玎：国内的很多程序员真的非常优秀。如果是 senior 的话，在英国的收入能达到怎样的水平?

董京：senior 的话，当然不能跟 Facebook 比，至少在 60000 英镑朝上。

叶玎玎：所以可以给国内一些人参考。现在人肉翻墙是一个很火的话题。

董京：特别是上海，上海这个天气，真的是没有办法形容。因为我是2月2日回上海的，那个飞机要迫降到厦门然后再到厦门，就是因为雾霾，整个飞机晚点了 6 个小时。

叶玎玎：但是伦敦比上海就少了一个霾，雾还是有的嘛。

董京：雾其实没有很多，但是雨很多，我在回上海的时候差不多一个月没有见过太阳。冬天一直在下雨。

叶玎玎：像 Qubit 来说，待了 3 年多，是家创业公司，之前你提到 F1，在英国电信有过工作经历，所以我想了解你在 F1 干嘛了?

董京：我之前在 F1 做了两个赛季，主要是帮他们写一些赛车策略上的软件。看赛车的话，国内应该还是有很多 F1 的 fans。从专业角度上看，F1 要看策略。看策略还是蛮有意思的。在欧洲有一些电视台，他们有一些策略的软件，能提前预测，赛车的油要加多少，轮胎要换什么样子，driver 是怎么开车的。每个赛车队都是有自己的一套软件的。

叶玎玎：你在哪个赛车队？

董京：我之前在雷诺，现在车队改名叫莲花。

叶玎玎：你相当于做策略分析?

董京：帮他们提供一些软件。做策略分析要专业的策略分析师。但我在层面上还是有一些了解的。每个车队都有一些历史数据，包括每个赛车手开车的习惯是什么，如 Lewis Hamilton 他打弯的时候是非常用力的，有一边的轮胎在磨损上比其他赛车手要高一些。数据接收的话，大部分赛车预测的数据都是实时给出。有一个公司叫 FOM（Formula One Management有限公司），他们提供整个比赛的规则和数据。大部分数据都是和时间有关，比如车子通过每一个赛道的时间是多少，他跑到第几圈。在赛外知道这件事情，还是非常 surprise 的。他们可以通过非常简单的一个时间数据就可以从头到尾了解到这个 driver 的 performance，包括其他车手的 performance。还是蛮有趣的一个工作。

叶玎玎：听起来是科技改变赛车。

董京：这有件搞笑的事。虽然看起来他很高端，但当年的时候，那些数据都是靠人工提供的。有一个人在赛道拿着一个计时器，每辆赛车跑过的时候他都会按一下计时器。但是现在会好一点。

叶玎玎：你有没上去玩过?

董京：我没有上去过。不过我之前的办公室是在雷诺的一个工厂里面，在一个很偏僻，没有人烟的地方。上班如果骑车或者开车，只能看到一些动物，比如狐狸，梅花鹿，总之就是很偏僻的小山村。之前雷诺的工场分了两个。一个是在英国，一个是在法国。法国主要是造引擎的。英国是造车架的，包括整个车子的外框、车头。就是前翼、后翼和车身。英国这里还包括提供一些软件更新。那时候我的办公室楼下就是组装的地方。每个车队都要准备 4 辆车，2 个给 driver，另外两台给车手赛前预备。

叶玎玎：想想都觉得酷。

董京：基本上每天下午都要试引擎。每天 5 点钟之后，在我办公室下面要开引擎热车，那声音非常非常响。他们加了消音器之后，声音还是非常非常响。

叶玎玎：不过从这段经历啊，到你后面的东西，都是在和数据打交道。

董京：都是在这个产业里做的。不知道怎么可以脱离这个行业。我对数据处理，怎么样让用户了解怎么去使用数据的了解还是比较多的。

叶玎玎：其实不用脱离啊，我觉得这块现在很好。大数据其实很火。那你在学校是学什么的呢?

董京：我是 Computer science 的。

叶玎玎：你是在计算机专业里偏数据这一块吗?

董京：偏理论一些。

叶玎玎：你在英国待了多久?

董京：差不多 10 个年头了吧。然后 5 年学习，5 年工作。

叶玎玎：相当于在英国读了个大学就开始工作。

董京：大学本科和硕士，然后再工作。

叶玎玎：英国本硕只要 5 年。

董京：本科是 4 年，硕士 1 年。

叶玎玎：硕士在英国只需要 1 年，还是你学的比较快?

董京：只要 1 年。我是在帝国理工攻读的硕士。在帝国理工，学硕士是非常折磨人的。帝国理工是一个蛮有名以工程著名的学校，毕业率是非常低。如果你学 EE，或者是 CS，基本上毕业率在 60%。我第一次接触 Ruby 的时候是在 2008 年，还是比较早的。我记得 08 年那时候，Ruby 的版本是 1.8，Rails 刚出到 2。我大学的一个学长，他是一个很不淡定的人。他毕业的时候，一直想做一个创业公司，一个 social net-work。他找到我，说，“那好吧，那我们要用什么来写？”他可能想写 PHP，而我又是一个 hipster 的人，想学一个新的语言。当时我就比较喜欢 Ruby，所以就很莫名其妙的看了这一套东西。所以说就帮他做了一点东西。但是他当时对这个东西不是特别感兴趣。所以我跟他就是因为兴趣不合，就分开来了，也祝福他能把东西做好。后来我在大学毕业论文的时候，

就写了一套 social network。我对这个东西还是蛮感兴趣的，就用 Rails 构架了一个 social network，有点像 Facebook，有点像 forum。像一个论坛，但是这个论坛里可以构架一些 apps。我记得当时在论文里写了一些关于 BDD 的东西。我还是蛮早的时候就接触了这一套东西。

叶玎玎：刚才你提到了一些 Node 不是特别舒服的地方，比如不太注重工程。

董京：我可能是片面的了解了。大部分写前端代码的人，没有太多的后端的经验。当然有些工程师有，但是他们没有太多后端的经历的话，对于一些具体的比较工程类的构架或者是怎么样能写出可以维护的代码不太了解。但都是因人而异。

叶玎玎：我们原来采访了朴灵嘛，朴灵也提到了类似的观点。把 Node 用得比较好的，都是来自后端的人。他们在后端有一些比较强的 background，到了前端这块，他们的整个思维会带过来。真正做得好的，能推进 Node 的可能会是后端的这些人。

董京：非常同意。

Part 4 - Share Picks

叶玎玎：今天非常感谢你这么早起来录这一期的 Teahour，今天的节目就到这里为止。下面就是我们 Teahour 的一个例行环节，叫 short picks。Short picks 就是你可以任意分享一个你觉得有意思的东西，或者想玩的东西，或者在看的一本书。你先来?

董京：好吧。我说的可能会稍微长一点。我的兴趣还是蛮多的。去年我大概花了 1 年的去玩多轴飞行器。我推荐大家去看看这个公司，叫做 3DR（3DRobotics），是 Chris Anderson 开的一个专门做飞行器的公司。最近这个主题还是蛮流行的。最近的新闻中 Facebook 要做他们自己的飞行器。而之前 Amazon 也有用他们自己的飞行器做一些货物的 delivery。

叶玎玎：自动送货。

董京：这一方面是一个趋势。我对这个方面也是蛮感兴趣的。大家可以看一下 3DR。和一个最新的飞行控制器的平台，叫 Pixhawk 。这个飞行控制器是一个非常好的 32 位飞行控制器。之前的都是 8 位的，而这个控制器可以做平行计算，可以加非常多的感光器件。

叶玎玎：没有图片，我没法想象。我看过一种，是 4 个圆组成的方，每一个圆里有一个机扇的那种。

董京：差不多吧。他是分多轴的嘛。你可以选 4 个，选 6 个。如果大家看《爸爸去哪儿》，他们拍摄的时候都是用那个拍的。他在空中拍摄都是用四轴拍的。

叶玎玎：你等一下可以给我一张你在玩那个的照片。

董京：我可以给一个链接给大家看我是怎么把它摔坏的录像。

叶玎玎：好的。这个我没玩过，我不太了解。但是我在外面看别人玩过，是挺好玩的。

董京：这个东西还算是蛮贵的。我有一个理想就是退休后读一个博士，往这个方向读。

叶玎玎：好吧。听你这么介绍，是可编程的吗?

董京：对，是可编程的。这个社区还是蛮大的。有不同版本的飞行控制器，有几个比较有名的，有一个叫 ??。有一个法国人，他把任天堂 Wii Remote 的 controller 全部拆掉，然后把它改成了一个飞控的平衡控制器。在另外一个社区 3DR，他们专门造的一个飞行控制器，叫 APM。也是一个非常有名的飞行控制器。最近他们发布的一个 Pixhawk，这个新的控制器可以用Lua编译代码。之前老的都是要写类似于 C 的代码，通过 Arduino 来编译的。

叶玎玎：什么时候让我看看你那个摔坏的视频能不能打动我，我也想尝试一下。

董京：玩这个要注意安全，还是蛮危险的。

叶玎玎：也就是说这个不太适合给小孩子当玩具。

董京：不太适合，这个东西不是属于玩具类的，是属于“杀伤性武器”。

叶玎玎：好的，OK。那我放弃这个想法。

董京：你可以买一些很简单的，还有一些玩具类的，不需要太多的调试就可以飞的那些。那些还是可以做玩具的。它有一些小的飞行控制器，大概有手掌那么大吧。还是可以给小孩子玩玩的。

叶玎玎：我觉得小孩子应该还是挺喜欢玩得。谢谢介绍。还有没有其他的?

董京：没有了。

叶玎玎：OK，那今天我的 picks 是一个软件。最近 Google 停止支持 Gmail Notifier 了。那我向大家推荐一个我朋友写的软件，叫 Gmail Notifr，是一个 open source 的在 Mac 上的客户端。很多人的工作习惯都会例行处理邮件。有的时候，我们看到一个邮件提醒，让我们知道大概什么东西发过来了。但是我们不会想实时的去检查，而是每隔一小时两小时，让它自动去检查一下，然后告诉我邮箱里有邮件。所以说用一个 Gmail Notifr，会比较简单的一点，让你觉得工作的更加高效。所以我就推荐一下我朋友写的，我自己在用的不错的软件，叫做—— Gmail Notifr。在 Mac App Store 可以下载。大家可以来支持一下，尝试一下。当然他也是开源的，你可以免费下载。

董京：那不错，我记得在回国的时候，由于我很多的邮件都是在 Gmail 上，访问起来非常慢。

叶玎玎：对。而且用 web 的话会更加慢，用客户端会好很多。IMAP 还好一点。OK，这就是我今天的 picks，最后非常感谢董京来到我们的 Tea-hour 做客，希望下一次有机会可以在跟你深入聊一下 Qubit 用到的东西。

董京：可以啊。然后还要告诉大家一下，希望大家期待正在计划组织下一次 Hacker News Meetup。

叶玎玎：顺便说一说，因为你现在正在招人。你可以留个邮箱，让大家可以直接联系你。我相信会有很多人听了这期节目会对加入你们的团队有兴趣。

董京：好的。或者大家可以在微博上@我，我就会看到。

叶玎玎：好，那就这样?

董京：好。谢谢。

叶玎玎：88

董京：88

原文链接:

http://yedingding.com/2014/05/04/teahour-50-with-jing.html

TCP协议疑难杂症全景解析

作者:nono

说明:
1).本文以TCP的发展历程解析容易引起混淆，误会的方方面面

2).本文不会贴大量的源码，大多数是以文字形式描述，我相信文字看起来是要比代码更轻松的

3).针对对象：对TCP已经有了全面了解的人。因为本文不会解析TCP头里面的每一个字段或者3次握手的细节，也不会解释慢启动和快速重传的定义

4).除了《TCP/IP详解》(卷一，卷二)以及《Unix网络编程》以及Linux源代码之外，学习网络更好的资源是RFC

5).本文给出一个提纲，如果想了解细节，请直接查阅RFC

6).翻来覆去，终于找到了这篇备忘，本文基于这篇备忘文档修改。

1.网络协议设计

ISO提出了OSI分层网络模型，这种分层模型是理论上的，TCP/IP最终实现了一个分层的协议模型，每一个层次对应一组网络协议完成一组特定的功能，该组网络协议被其下的层次复用和解复用。这就是分层模型的本质，最终所有的逻辑被编码到线缆或者电磁波。

分层模型是很好理解的，然而对于每一层的协议设计却不是那么容易。TCP/IP的漂亮之处在于：协议越往上层越复杂。我们把网络定义为互相连接在一起的设备，网络的本质作用还是“端到端”的通信，然而希望互相通信的设备并不一定要“直接”连接在一起，因此必然需要一些中间的设备负责转发数据，因此就把连接这些中间设备的线缆上跑的协议定义为链路层协议，

实际上所谓链路其实就是始发与一个设备，通过一根线，终止于另一个设备。我们把一条链路称为“一跳”。因此一个端到端的网络包含了“很多跳”。

2.TCP和IP协议

终止于IP协议，我们已经可以完成一个端到端的通信，为何还需要TCP协议？这是一个问题，理解了这个问题，我们就能理解TCP协议为何成了现在这个样子，为何如此“复杂”，为何又如此简单。

正如其名字所展示的那样，TCP的作用是传输控制，也就是控制端到端的传输，那为何这种控制不在IP协议中实现的。答案很简单，那就是这会增加IP协议的复杂性，而IP协议需要的就是简单。这是什么原因造成的呢？

首先我们认识一下为何IP协议是沙漏的细腰部分。它的下层是繁多的链路层协议，这些链路提供了相互截然不同且相差很远的语义，为了互联这些异构的网络，我们需要一个网络层协议起码要提供一些适配的功能，另外它必然不能提供太多的“保证性服务”，因为上层的保证性依赖下层的约束性更强的保证性，你永远无法在一个100M吞吐量的链路之上实现的IP协议保证1000M的吞吐量...

IP协议设计为分组转发协议，每一跳都要经过一个中间节点，路由的设计是TCP/IP网络的另一大创举，这样，IP协议就无需方向性，路由信息和协议本身不再强关联，它们仅仅通过IP地址来关联，因此，IP协议更加简单。路由器作为中间节点也不能太复杂，这涉及到成本问题，因此路由器只负责选路以及转发数据包。

因此传输控制协议必然需要在端点实现。在我们详谈TCP协议之前，首先要看一下它不能做什么，由于IP协议不提供保证，TCP也不能提供依赖于IP下层链路的这种保证，比如带宽，比如时延，这些都是链路层决定的，既然IP协议无法修补，TCP也不能，然而它却能修正始于IP层的一些“不可保证性质”，这些性质包括IP层的不可靠，IP层的不按顺序，IP层的无方向/无连接。

将该小节总结一下，TCP/IP模型从下往上，功能增加，需要实现的设备减少，然而设备的复杂性却在增加，这样保证了成本的最小化，至于性能或者因素，靠软件来调节吧，TCP协议就是这样的软件，实际上最开始的时

候，TCP并不考虑性能，效率，公平性，正是考虑了这些，TCP协议才复杂了起来。

3.TCP协议

这是一个纯软件协议，为何将其设计上两个端点，参见上一小节，本节详述TCP协议，中间也穿插一些简短的论述。

3.1.TCP协议

确切的说，TCP协议有两重身份，作为网络协议，它弥补了IP协议尽力而为服务的不足，实现了有连接，可靠传输，报文按序到达。作为一个主机软件，它和UDP以及左右的传输层协议隔离了主机服务和网络，它们可以被看做是一个多路复用/解复用器，将诸多的主机进程数据复用/解复用到IP层。可以看出，不管从哪个角度，TCP都作为一个接口存在，作为网络协议，它和对端的TCP接口，实现TCP的控制逻辑，作为多路复用/解复用器，它和下层IP协议接口，实现协议栈的功能，而这正是分层网络协议模型的基本定义(两类接口，一类和下层接口，另一类和对等层接口)。

我们习惯于将TCP作为协议栈的最顶端，而不把应用层协议当成协议栈的一部分，这部分是因为应用层被TCP/UDP解复用了之后，呈现出了一种太复杂的局面，应用层协议用一种不同截然不同的方式被解释，应用层协议习惯于用类似ASN.1标准来封装，这正体现了TCP协议作为多路复用/解复用器的重要性，由于直接和应用接口，它可以很容易直接被应用控制，实现不同的传输控制策略，这也是TCP被设计到离应用不太远的地方的原因之一。

总之，TCP要点有四，一曰有连接，二曰可靠传输，三曰数据按照到达，四曰端到端流量控制。注意，TCP被设计时只保证这四点，此时它虽然也有些问题，然而很简单，然而更大的问题很快呈现出来，使之不得不考虑和IP网络相关的东西，比如公平性，效率，因此增加了拥塞控制，这样TCP就成了现在这个样子。

3.2.有连接，可靠传输，数据按序到达的TCP

IP协议是没有方向的，数据报传输能到达对端全靠路由，因此它是一跳一跳地到达对端的，只要有一跳没有到达对端的路由，那么数据传输将失败，其实路由也是互联网的核心之一，实际上IP层提供的核心基本功能有两点，第

一点是地址管理，第二点就是路由选路。TCP利用了IP路由这个简单的功能，因此TCP不必考虑选路，这又一个它被设计成端到端协议的原因。

既然IP已经能尽力让单独的数据报到达对端，那么TCP就可以在这种尽力而为的网络上实现其它的更加严格的控制功能。TCP给无连接的IP网络通信增加了连接性，确认了已经发送出去的数据的状态，并且保证了数据的顺序。

3.2.1.有连接

这是TCP的基本，因为后续的传输的可靠性以及数据顺序性都依赖于一条连接，这是最简单的实现方式，因此TCP被设计成一种基于流的协议，既然TCP需要事先建立连接，之后传输多少数据就无所谓了，只要是同一连接的数据能识别出来即可。

疑难杂症1：3次握手和4次挥手

TCP使用3次握手建立一条连接，该握手初始化了传输可靠性以及数据顺序性必要的信息，这些信息包括两个方向的初始序列号，确认号由初始序列号生成，使用3次握手是因为3次握手已经准备好了传输可靠性以及数据顺序性所必要的信息，该握手的第3次实际上并不是需要单独传输的，完全可以和数据一起传输。

TCP使用4次挥手拆除一条连接，为何需要4次呢？因为TCP是一个全双工协议，必须单独拆除每一条信道。注意，4次挥手和3次握手的意义是不同的，很多人都会问为何建立连接是3次握手，而拆除连接是4次挥手。3次握手的目的很简单，就是分配资源，初始化序列号，这时还不涉及数据传输，3次就足够做到这个了，而4次挥手的目的是终止数据传输，并回收资源，此时两个端点两个方向的序列号已经没有了任何关系，必须等待两方向都没有数据传输时才能拆除虚链路，不像初始化时那么简单，发现SYN标志就初始化一个序列号并确认SYN的序列号。因此必须单独分别在一个方向上终止该方向的数据传输。

疑难杂症2：TIME_WAIT状态

为何要有这个状态，原因很简单，那就是每次建立连接的时候序列号都是随机产生的，并且这个序列号是32位的，会回绕。现在我来解释这和 TIME_WAIT 有什么关系。

任何的TCP分段都要在尽力而为的IP网络上传输，中间的路由器可能会随意的缓存任何的IP数据报，它并不管这个IP数据报上被承载的是什么数据，然而根据经验和互联网的大小，一个IP数据报最多存活MSL(这是根据地球表面积，电磁波在各种介质中的传输速率以及IP协议的TTL等综合推算出来的，如果在火星上，这个MSL会大得多...)。

现在我们考虑终止连接时的被动方发送了一个FIN，然后主动方回复了一个ACK，然而这个ACK可能会丢失，这会造成被动方重发FIN，这个FIN可能会在互联网上存活MSL。

如果没有TIME_WAIT的话，假设连接1已经断开，然而其被动方最后重发的那个FIN(或者FIN之前发送的任何TCP分段)还在网络上，然而连接2重用了连接1的所有的5元素(源IP，目的IP，TCP，源端口，目的端口)，刚刚将建立好连接，连接1迟到的FIN到达了，这个FIN将以比较低但是确实可能的概率终止掉连接2.

为何说是概率比较低呢？这涉及到一个匹配问题，迟到的FIN分段的序列号必须落在连接2的一方的期望序列号范围之内。虽然这种巧合很少发生，但确实会发生，毕竟初始序列号是随机产生了。因此终止连接的主动方必须在接受了被动方且回复了ACK之后等待2*MSL时间才能进入CLOSE状态，之所以乘以2是因为这是保守的算法，最坏情况下，针对被动方的ACK在以最长路线(经历一个MSL)经过互联网马上到达被动方时丢失。

为了应对这个问题，RFC793对初始序列号的生成有个建议，那就是设定一个基准，在这个基准之上搞随机，这个基准就是时间，我们知道时间是单调递增的。然而这仍然有问题，那就是回绕问题，如果发生回绕，那么新的序列号将会落到一个很低的值。因此最好的办法就是避开“重叠”，其含义就是基准之上的随机要设定一个范围。

要知道，很多人很不喜欢看到服务器上出现大量的TIME_WAIT状态的连接，因此他们将TIME_WAIT的值设置的很低，这虽然在大多数情况下可行，然而确实也是一种冒险行为。最好的方式就是，不要重用一个连接。

疑难杂症3：重用一个连接和重用一个套接字

这是根本不同的，单独重用一个套接字一般不会有任何问题，因为TCP是基于连接的。比如在服务器端出现了一个TIME_WAIT连接，那么该连接标识了一个五元素，只要客户端不使用相同的源端口，连接服务器是没有问题

的，因为迟到的FIN永远不会到达这个连接。记住，一个五元素标识了一个连接，而不是一个套接字(当然，对于BSD套接字而言，服务端的accept套接字确实标识了一个连接)。

3.2.2.传输可靠性

基本上传输可靠性是靠确认号实现的，也就是说，每发送一个分段，接下来接收端必然要发送一个确认，发送端收到确认后才可以发送下一个字节。这个原则最简单不过了，教科书上的“停止-等待”协议就是这个原则的字节版本，只是TCP使用了滑动窗口机制使得每次不一定发送一个字节，但是这是后话，本节仅仅谈一下确认的超时机制。

怎么知道数据到达对端呢？那就是对端发送一个确认，但是如果一直收不到对端的确认，发送端等多久呢？如果一直等下去，那么将无法发现数据的丢失，协议将不可用，如果等待时间过短，可能确认还在路上，因此等待时间是个问题，另外如何去管理这个超时时间也是一个问题。

疑难杂症4：超时时间的计算

绝对不能随意去揣测超时的时间，而应该给出一个精确的算法去计算。毫无疑问，一个TCP分段的回复到达的时间就是一个数据报往返的时间，因此标准定义了一个新的名词RTT，代表一个TCP分段的往返时间。然而我们知道，IP网络是尽力而为的，并且路由是动态的，且路由器会毫无先兆的缓存或者丢弃任何的数据报，因此这个RTT是需要动态测量的，也就是说起码每隔一段时间就要测量一次，如果每次都一样，万事大吉，然而世界并非如你所愿，因此我们需要找到的恰恰的一个“平均值”，而不是一个准确值。

这个平均值如果仅仅直接通过计算多次测量值取算术平均，那是不恰当的，因为对于数据传输延时，我们必须考虑的路径延迟的瞬间抖动，否则如果两次测量值分别为2和98，那么超时值将是50，这个值对于2而言，太大了，结果造成了数据的延迟过大(本该重传的等待了好久才重传)，然而对于98而言，太小了，结果造成了过度重传(路途遥远，本该很慢，结果大量重传已经正确确认但是迟到的TCP分段)。

因此，除了考虑每两次测量值的偏差之外，其变化率也应该考虑在内，如果变化率过大，则通过以变化率为自变量的函数为主计算RTT(如果陡然增大，则取值为比较大的正数，如果陡然减小，则取值为比较小的负数，然后和平均值加权求和)，反之如果变化率很小，则取测量平均值。这是不言而喻的，这个算法至今仍然工作的很好。

疑难杂症5：超时计时器的管理-每连接单一计时器

很显然，对每一个TCP分段都生成一个计时器是最直接的方式，每个计时器在RTT时间后到期，如果没有收到确认，则重传。然而这只是理论上的合理，对于大多数操作系统而言，这将带来巨大的内存开销和调度开销，因此采取每一个TCP连接单一计时器的设计则成了一个默认的选择。可是单一的计时器怎么管理如此多的发出去的TCP分段呢？又该如何来设计单一的计时器呢。

设计单一计时器有两个原则：1.每一个报文在长期收不到确认都必须可以超时；2.这个长期收不到中长期不能和测量的RTT相隔太远。因此RFC2988定义一套很简单的原则：

a.发送TCP分段时，如果还没有重传定时器开启，那么开启它。
b.发送TCP分段时，如果已经有重传定时器开启，不再开启它。
c.收到一个非冗余ACK时，如果有数据在传输中，重新开启重传定时器。
d.收到一个非冗余ACK时，如果没有数据在传输中，则关闭重传定时器。

我们看看这4条规则是如何做到以上两点的，根据a和c(在c中，注意到ACK是非冗余的)，任何TCP分段只要不被确认，超时定时器总会超时的。然而为何需要c呢？只有规则a存在的话，也可以做到原则1。实际上确实是这样的，但是为了不会出现过早重传，才添加了规则c，如果没有规则c，那么万一在重传定时器到期前，发送了一些数据，这样在定时器到期后，除了很早发送的数据能收到ACK外，其它稍晚些发送的数据的ACK都将不会到来，因此这些数据都将被重传。有了规则c之后，只要有分段ACK到来，则重置重传定时器，这很合理，因此大多数正常情况下，从数据的发出到ACK的到来这段时间以及计算得到的RTT以及重传定时器超时的时间这三者相差并不大，一个ACK到来后重置定时器可以保护后发的数据不被过早重传。

这里面还有一些细节需要说明。一个ACK到来了，说明后续的ACK很可能会依次到来，也就是说丢失的可能性并不大，另外，即使真的有后发的TCP分段丢失现象发生，也会在最多2倍定时器超时时间的范围内被重传(假设该报文是第一个报文发出启动定时器之后马上发出的，丢失了，第一个报文的ACK到来后又重启了定时器，又经过了一个超时时间才会被重传)。虽然这里还没有涉及拥塞控制，但是可见网络拥塞会引起丢包，丢包会引起重传，过度重传反过来加重网络拥塞，设置规则c的结果可以缓解过多的重传，毕竟将启动定时器之后发送的数据的重传超时时间拉长了最多一倍左

右。最多一倍左右的超时偏差做到了原则2，即“这个长期收不到中长期不能和测量的RTT相隔太远”。

还有一点，如果是一个发送序列的最后一个分段丢失了，后面就不会收到冗余ACK，这样就只能等到超时了，并且超时时间几乎是肯定会比定时器超时时间更长。如果这个分段是在发送序列的靠后的时间发送的且和前面的发送时间相隔时间较远，则其超时时间不会很大，反之就会比较大。

疑难杂症6：何时测量RTT

目前很多TCP实现了时间戳，这样就方便多了，发送端再也不需要保存发送分段的时间了，只需要将其放入协议头的时间戳字段，然后接收端将其回显在ACK即可，然后发送端收到ACK后，取出时间戳，和当前时间做算术差，即可完成一次RTT的测量。

3.2.3.数据顺序性

基本上传输可靠性是靠序列号实现的。

疑难杂症7：确认号和超时重传

确认号是一个很诡异的东西，因为TCP的发送端对于发送出去的一个数据序列，它只要收到一个确认号就认为确认号前面的数据都被收到了，即使前面的某个确认号丢失了，也就是说，发送端只认最后一个确认号。这是合理的，因为确认号是接收端发出的，接收端只确认按序到达的最后一个TCP分段。

另外，发送端重发了一个TCP报文并且接收到该TCP分段的确认号，并不能说明这个重发的报文被接收了，也可能是数据早就被接收了，只是由于其ACK丢失或者其ACK延迟到达导致了超时。值得说明的是，接收端会丢弃任何重复的数据，即使丢弃了重复的数据，其ACK还是会照发不误的。

标准的早期TCP实现为，只要一个TCP分段丢失，即使后面的TCP分段都被完整收到，发送端还是会重传从丢失分段开始的所有报文，这就会导致一个问题，那就是重传风暴，一个分段丢失，引起大量的重传。这种风暴实则不必要的，因为大多数的TCP实现中，接收端已经缓存了乱序的分段，这些被重传的丢失分段之后的分段到达接收端之后，很大的可能性是被丢弃。

关于这一点在拥塞控制被引入之后还会提及(问题先述为快：本来报文丢失导致超时就说明网络很可能已然拥塞，重传风暴只能加重其拥塞程度)。

疑难杂症8：乱序数据缓存以及选择确认

TCP是保证数据顺序的，但是并不意味着它总是会丢弃乱序的TCP分段，具体会不会丢弃是和具体实现相关的，RFC建议如果内存允许，还是要缓存这些乱序到来的分段，然后实现一种机制等到可以拼接成一个按序序列的时候将缓存的分段拼接，这就类似于IP协议中的分片一样，但是由于IP数据报是不确认的，因此IP协议的实现必须缓存收到的任何分片而不能将其丢弃，因为丢弃了一个IP分片，它就再也不会到来了。

现在，TCP实现了一种称为选择确认的方式，接收端会显式告诉发送端需要重传哪些分段而不需要重传哪些分段。这无疑避免了重传风暴。

疑难杂症9：TCP序列号的回绕的问题

TCP的序列号回绕会引起很多的问题，比如序列号为s的分段发出之后，m秒后，序列号比s小的序列号为j的分段发出，只不过此时的j比上一个s多了一圈，这就是回绕问题，那么如果这后一个分段到达接收端，这就会引发彻底乱序-本来j该在s后面，结果反而到达前面了，这种乱序是TCP协议检查不出来的。我们仔细想一下，这种情况确实会发生，数据分段并不是一个字节一个字节发送出去的，如果存在一个速率为1Gbps的网络，TCP发送端1秒会发送125MB的数据，32位的序列号空间能传输2的32次方个字节，也就是说32秒左右就会发生回绕，我们知道这个值远小于MSL值，因此会发生的。

有个细节可能会引起误会，那就是TCP的窗口大小空间是序列号空间的一半，这样恰好在满载情况下，数据能填满发送窗口和接收窗口，序列号空间正好够用。然而事实上，TCP的初始序列号并不是从0开始的，而是随机产生的(当然要辅助一些更精妙的算法)，因此如果初始序列号比较接近2的32次方，那么很快就会回绕。

当然，如今可以用时间戳选项来辅助作为序列号的一个识别的部分，接收端遇到回绕的情况，需要比较时间戳，我们知道，时间戳是单调递增的，虽然也会回绕，然而回绕时间却要长很多。这只是一种策略，在此不详谈。还有一个很现实的问题，理论上序列号会回绕，但是实际上，有多少TCP的端点主机直接架设在1G的网络线缆两端并且接收方和发送方的窗口还能恰好被同时填满。另外，就算发生了回绕，也不是一件特别的事情，回绕在计

算机里面太常见了，只需要能识别出来即可解决，对于TCP的序列号而言，在高速网络(点对点网络或者以太网)的两端，数据发生乱序的可能性很小，因此当收到一个序列号突然变为0或者终止序列号小于起始序列号的情况后，很容易辨别出来，只需要和前一个确认的分段比较即可，如果在一个经过路由器的网络两端，会引发IP数据报的顺序重排，对于TCP而言，虽然还会发生回绕，也会慢得多，且考虑到拥塞窗口(目前还没有引入)一般不会太大，窗口也很难被填满到65536。

3.2.4.端到端的流量控制

端到端的流量控制使用滑动窗口来实现。滑动窗口的原理非常简单，基本就是一个生产者/消费者模型

疑难杂症10：流量控制的真实意义

很多人以为流量控制会很有效的协调两端的流量匹配，确实是这样，但是如果你考虑到网络的利用率问题，TCP的流量控制机制就不那么完美了，造成这种局面的原因在于，滑动窗口只是限制了最大发送的数据，却没有限制最小发送的数据，结果导致一些很小的数据被封装成TCP分段，报文协议头所占的比例过于大，造成网络利用率下降，这就引出了接下来的内容，那就是端到端意义的TCP协议效率。

~~~~~~~~~~~~~~~~~~~~~~
承上启下
终于到了阐述问题的时候了，以上的**TCP**协议实现的非常简单，这也是**TCP**的标准实现，然而很快我们就会发现各种各样的问题。这些问题导致了标准化协会对**TCP**协议进行了大量的修补，这些修补杂糅在一起让人们有些云里雾里，不知所措。本文档就旨在分离这些杂乱的情况，实际上，根据**RFC**，这些杂乱的情况都是可以找到其单独的发展轨迹的。
~~~~~~~~~~~~~~~~~~~~~~

4.端到端意义上的TCP协议效率

4.1.三个问题以及解决

问题1描述：接收端处理慢，导致接收窗口被填满

这明显是速率不匹配引发的问题，然而即使速率不匹配，只要滑动窗口能协调好它们的速率就好，要快都快，要慢都慢，事实上滑动窗口在这一点上做的很好。但是如果我们不得不从效率上来考虑问题的话，事实就不那么乐观了。考虑此时接收窗口已然被填满，慢速的应用程序慢腾腾的读取了一个字节，空出一个位置，然后通告给TCP的发送端，发送端得知空出一个位置，马上发出一个字节，又将接收端填满，然后接收应用程序又一次慢腾腾...这就是糊涂窗口综合症，一个大多数人都很熟悉的词。这个问题极大的浪费了网络带宽，降低了网络利用率。好比从大同拉100吨煤到北京需要一辆车，拉1Kg煤到北京也需要一辆车(超级夸张的一个例子，请不要相信)，但是一辆车开到北京的开销是一定的...

问题1解决：窗口通告

对于问题1，很显然问题出在接收端，我们没有办法限制发送端不发送小分段，但是却可以限制接收端通告小窗口，这是合理的，这并不影响应用程序，此时经典的延迟/吞吐量反比律将不再适用，因为接收窗口是满的，其空出一半空间表示还有一半空间有数据没有被应用读取，和其空出一个字节的空间的效果是一样的，因此可以限制接收端当窗口为0时，直接通告给发送端以阻止其继续发送数据，只有当其接收窗口再次达到MSS的一半大小的时候才通告一个不为0的窗口，此前对于所有的发送端的窗口probe分段(用于探测接收端窗口大小的probe分段，由TCP标准规定)，全部通告窗口为0，这样发送端在收到窗口不为0的通告，那么肯定是一个比较大的窗口，因此发送端可以一次性发出一个很大的TCP分段，包含大量数据，也即拉了好几十吨的煤到北京，而不是只拉了几公斤。

即，限制窗口通告时机，解决糊涂窗口综合症

问题2描述：发送端持续发送小包，导致窗口闲置

这明显是发送端引起的问题，此时接收端的窗口开得很大，然而发送端却不积累数据，还是一味的发送小块数据分段。只要发送了任和的分段，接收端都要无条件接收并且确认，这完全符合TCP规范，因此必然要限制发送端不发送这样的小分段。

问题2解决：Nagle算法

Nagel算法很简单，标准的Nagle算法为：

IF 数据的大小和窗口的大小都超过了MSS
Then 发送数据分段
ELSE

IF 还有发出的TCP分段的确认没有到来
Then 积累数据到发送队列的末尾的TCP分段
ELSE
发送数据分段
EndIF
EndIF

可是后来，这个算法变了，变得更加灵活了，其中的：

IF 还有发出的TCP分段的确认没有到来

变成了

IF 还有发出的不足MSS大小的TCP分段的确认没有到来

这样如果发出了一个MSS大小的分段还没有被确认，后面也是可以随时发送一个小分段的，这个改进降低了算法对延迟时间的影响。这个算法体现了一种自适应的策略，越是确认的快，越是发送的快，虽然Nagle算法看起来在积累数据增加吞吐量的同时也加大的时延，可事实上，如果对于类似交互式的应用，时延并不会增加，因为这类应用回复数据也是很快的，比如Telnet之类的服务必然需要回显字符，因此能和对端进行自适应协调。

注意，Nagle算法是默认开启的，但是却可以关闭。如果在开启的情况下，那么它就严格按照上述的算法来执行。

问题3.确认号(ACK)本身就是不含数据的分段，因此大量的确认号消耗了大量的带宽

这是TCP为了确保可靠性传输的规范，然而大多数情况下，ACK还是可以和数据一起捎带传输的。如果没有捎带传输，那么就只能单独回来一个ACK，如果这样的分段太多，网络的利用率就会下降。从大同用火车拉到北京100吨煤，为了确认煤已收到，北京需要派一辆同样的火车空载开到大同去复命，因为没有别的交通工具，只有火车。如果这位复命者刚开着一列火车走，又从大同来了一车煤，这拉煤的哥们儿又要开一列空车去复命了。

问题3的解决：

RFC建议了一种延迟的ACK，也就是说，ACK在收到数据后并不马上回复，而是延迟一段可以接受的时间，延迟一段时间的目的是看能不能和接收方要发给发送方的数据一起回去，因为TCP协议头中总是包含确认号的，如果能的话，就将ACK一起捎带回去，这样网络利用率就提高了。往大同复命的确认者不必开一辆空载火车回大同了，此时北京正好有一批货物要送往大同，这位复命者搭着这批货的火车返回大同。

如果等了一段可以接受的时间，还是没有数据要发往发送端，此时就需要单独发送一个ACK了，然而即使如此，这个延迟的ACK虽然没有等到可以被捎带的数据分段，也可能等到了后续到来的TCP分段，这样它们就可以取最大者一起返回了，要知道，TCP的确认号是收到的按序报文的最后一个字节的后一个字节。最后，RFC建议，延迟的ACK最多等待两个分段的积累确认。

4.2.分析三个问题之间的关联

三个问题导致的结果是相同的，但是要知道它们的原因本质上是不同的，问题1几乎总是出现在接收端窗口满的情况下，而问题2几乎总是发生在窗口闲置的情况下，问题3看起来是最无聊的，然而由于TCP的要求，必须要有确认号，而且一个确认号就需要一个TCP分段，这个分段不含数据，无疑是很小的。

三个问题都导致了网络利用率的降低。虽然两个问题导致了同样的结果，但是必须认识到它们是不同的问题，很自然的将这些问题的解决方案汇总在一起，形成一个全局的解决方案，这就是如今的操作系统中的解决方案。

4.3.问题的杂糅情况

疑难杂症11：糊涂窗口解决方案和Nagle算法

糊涂窗口综合症患者希望发送端积累TCP分段，而Nagle算法确实保证了一定的TCP分段在发送端的积累，另外在延迟ACK的延迟的那一会时间，发送端会利用这段时间积累数据。然而这却是三个不同的问题。Nagle算法可以缓解糊涂窗口综合症，却不是治本的良药。

疑难杂症12：Nagle算法和延迟ACK

延迟ACK会延长ACK到达发送端的时间，由于标准Nagle算法只允许一个未被确认的TCP分段，那无疑在接收端，这个延迟的ACK是毫无希望等待后续数据到来最终进行积累确认的，如果没有数据可以捎带这个ACK，那么这个ACK只有在延迟确认定时器超时的时候才会发出，这样在等待这个ACK的过程中，发送端又积累了一些数据，因此延迟ACK实际上是在增加延迟的代价下加强了Nagle算法。在延迟ACK加Nagle算法的情况下，接收

端只有不断有数据要发回，才能同时既保证了发送端的分段积累，又保证了延迟不增加，同时还没有或者很少有空载的ACK。

要知道，延迟ACK和Nagle是两个问题的解决方案。

疑难杂症13：到底何时可以发送数据

到底何时才能发送数据呢？如果单从Nagle算法上看，很简单，然而事实证明，情况还要更复杂些。如果发送端已经排列了3个TCP分段，分段1，分段2，分段3依次被排入，三个分段都是小分段(不符合Nagle算法中立即发送的标准)，此时已经有一个分段被发出了，且其确认还没有到来，请问此时能发送分段1和2吗？如果按照Nagle算法，是不能发送的，但实际上它们是可以发送的，因为这两个分段已经没有任何机会再积累新的数据了，新的数据肯定都积累在分段3上了。问题在于，分段还没有积累到一定大小时，怎么还可以产生新的分段？这是可能的，但这是另一个问题，在此不谈。

Linux的TCP实现在这个问题上表现的更加灵活，它是这么判断能否发送的(在开启了Nagle的情况下)：

```
IF (没有超过拥塞窗口大小的数据分段未确认 || 数据分段中包含FIN ) &&
   数据分段没有超越窗口边界
   Then
   IF 分段在中间(上述例子中的分段1和2) ||
        分段是紧急模式          ||
     通过上述的Nagle算法(改进后的Nagle算法)
      Then 发送分段
   EndIF
EndIF
```

曾经我也改过Nagle算法，确切的说不是修改Nagle算法，而是修改了“到底何时能发送数据”的策略，以往都是发送端判断能否发送数据的，可是如果此时有延迟ACK在等待被捎带，而待发送的数据又由于积累不够或者其它原因不能发送，因此两边都在等，这其实在某些情况下不是很好。我所做的改进中对待何时能发送数据又增加了一种情况，这就是“ACK拉”的情况，一旦有延迟ACK等待发送，判断一下有没有数据也在等待发送，如果有的话，看看数据是否大到了一定程度，在此，我选择的是MSS的一半：

```
IF (没有超过拥塞窗口大小的数据分段未确认 || 数据分段中包含FIN ) &&
    数据分段没有超越窗口边界
```

```
    Then
    IF 分段在中间(上述例子中的分段1和2) ||
          分段是紧急模式            ||
       通过上述的Nagle算法(改进后的Nagle算法)
        Then 发送分段
    EndIF
ELSE IF 有延迟ACK等待传输                &&
    发送队列中有待发送的TCP分段        &&
    发送队列的头分段大小大于MSS的一半
       Then 发送队列头分段且捎带延迟ACK
EndIF
```

另外，发送队列头分段的大小是可以在统计意义上动态计算的，也不一定非要是MSS大小的一半。我们发现，这种算法对于交互式网路应用是自适应的，你打字越快，特定时间内积累的分段就越长，对端回复的越快(可以捎带ACK)，本端发送的也就越快(以Echo举例会更好理解)。

疑难杂症14：《TCP/IP详解(卷一)》中Nagle算法的例子解读

这个问题在网上搜了很多的答案，有的说RFC的建议，有的说别的。可是实际上这就是一个典型的“竞态问题”：
首先服务器发了两个分段：
数据段12：ack 14
数据段13：ack 14，54:56
然后客户端发了两个分段：
数据段14：ack 54，14:17
数据段15：ack 56，17:18
可以看到数据段14本来应该确认56的，但是确认的却是54。也就是说，数据段已经移出队列将要发送但还未发送的时候，数据段13才到来，软中断处理程序抢占了数据段14的发送进程，要知道此时只是把数据段14移出了队列，还没有更新任何的状态信息，比如“发出但未被确认的分段数量”，此时软中断处理程序顺利接收了分段13，然后更新窗口信息，并且检查看有没有数据要发送，由于分段14已经移出队列，下一个接受发送检查的就是分段15了，由于状态信息还没有更新，因此分段15顺利通过发送检测，发送完成。

可以看Linux的源代码了解相关信息，tcp_write_xmit这个函数在两个地方会被调用，一个是TCP的发送进程中，另一个就是软中断的接收处理中，两者在调用中的竞态就会引起《详解》中的那种情况。注意，这种不加锁的

发送方式是合理的，也是最高效的，因此TCP的处理语义会做出判断，丢弃一切不该接收或者重复接收的分段的。

~~~~~~~~~~~~~~~~~~~~~~
承上启下
又到了该承上启下，到此为止，我们叙述的**TCP**还都是简单的**TCP**，就算是简单的**TCP**，也存在上述的诸多问题，就更别提继续增加**TCP**的复杂性了。到此为止，我们的**TCP**都是端到端意义上的，然而实际上**TCP**要跑在**IP**网络之上的，而**IP**网络的问题是很多的，是一个很拥堵网络。不幸的是，**TCP**的有些关于确认和可靠性的机制还会加重**IP**网络的拥堵。
~~~~~~~~~~~~~~~~~~~~~~

5.IP网络之上的TCP

5.1.端到端的TCP协议和IP协议之间的矛盾

端到端的TCP只能看到两个节点，那就是自己和对方，它们是看不到任何中间的路径的。可是IP网络却是一跳一跳的，它们的矛盾之处在于TCP的端到端流量控制必然会导致网络拥堵。因为每条TCP连接的一端只知道它对端还有多少空间用于接收数据，它们并不管到达对端的路径上是否还有这么大的容量，事实上所有连接的这些空间加在一起将瞬间超过IP网络的容量，因此TCP也不可能按照滑动窗口流量控制机制很理想的运行。

势必需要一种拥塞控制机制，反应路径的拥塞情况。

疑难杂症15：拥塞控制的本质

由于TCP是端到端协议，因此两端之间的控制范畴属于流量控制，IP网络的拥塞会导致TCP分段的丢失，由于TCP看不到中间的路由器，因此这种丢失只会发生中间路由器，当然两个端点的网卡或者IP层丢掉数据分段也是TCP看不到的。因此拥塞控制必然作用于IP链路。事实上我们可以得知，只有在以下情况下拥塞控制才会起作用：

a.两个或两个以上的连接(其中一个一定要是TCP，另一个可以是任意连接)经过同一个路由器或者同一个链路时；
b.只有一个TCP连接，然而它经过了一个路由器时。

其它情况下是不会拥塞的。因为一个TCP总是希望独享整条网络通路，而这对于多个连接而言是不可能的，必须保证TCP的公平性，这样这种拥塞控制机制才合理。本质上，拥塞的原因就是大家都想独享全部带宽资源，结果导致拥塞，这也是合理的，毕竟TCP看不到网络的状态，同时这也决定了TCP的拥塞控制必须采用试探性的方式，最终到达一个足以引起其“反应”的“刺激点”。

拥塞控制需要完成以下两个任务：1.公平性；2.拥塞之后退出拥塞状态。

疑难杂症16：影响拥塞的因素

我们必须认识到拥塞控制是一个整体的机制，它不偏向于任何TCP连接，因此这个机制内在的就包含了公平性。那么影响拥塞的因素都有什么呢？具有讽刺意味的是，起初TCP并没有拥塞控制机制，正是TCP的超时重传风暴(一个分段丢失造成后续的已经发送的分段均被重传，而这些重传大多数是不必要的)加重了网络的拥塞。因此重传必然不能过频，必须把重传定时器的超时时间设置的稍微长一些，而这一点在单一重传定时器的设计中得到了加强。除此TCP自身的因素之外，其它所有的拥塞都可以靠拥塞控制机制来自动完成。

另外，不要把路由器想成一种线速转发设备，再好的路由器只要接入网络，总是会拉低网络的总带宽，因此即使只有一个TCP连接，由于TCP的发送方总是以发送链路的带宽发送分段，这些分段在经过路由器的时候排队和处理总是会有时延，因此最终肯定会丢包的。

最后，丢包的延后性也会加重拥塞。假设一个TCP连接经过了N个路由器，前N-1个路由器都能顺利转发TCP分段，但是最后一个路由器丢失了一个分段，这就导致了这些丢失的分段浪费了前面路由器的大量带宽。

5.2.拥塞控制的策略

在介绍拥塞控制之前，首先介绍一下拥塞窗口，它实际上表示的也是“可以发送多少数据”，然而这个和接收端通告的接收窗口意义是不一样的，后者是流量控制用的窗口，而前者是拥塞控制用的窗口，体现了网络拥塞程度。

拥塞控制整体上分为两类，一类是试探性的拥塞探测，另一类则是拥塞避免(注意，不是常规意义上的拥塞避免)。

5.2.1.试探性的拥塞探测分为两类，之一是慢启动，之二是拥塞窗口加性扩大(也就是熟知的拥塞避免，然而这种方式是避免不了拥塞的)。

5.2.2.拥塞避免方式拥塞控制旨在还没有发生拥塞的时候就先提醒发送端，网络拥塞了，这样发送端就要么可以进入快速重传/快速恢复或者显式的减小拥塞窗口，这样就避免网络拥塞的一沓糊涂之后出现超时，从而进入慢启动阶段。

5.2.3.快速重传和快速恢复。所谓快速重传/快速恢复是针对慢启动的，我们知道慢启动要从1个MSS开始增加拥塞窗口，而快速重传/快速恢复则是一旦收到3个冗余ACK，不必进入慢启动，而是将拥塞窗口缩小为当前阀值的一半加上3，然后如果继续收到冗余ACK，则将拥塞窗口加1个MSS，直到收到一个新的数据ACK，将窗口设置成正常的阀值，开始加性增加的阶段。

当进入快速重传时，为何要将拥塞窗口缩小为当前阀值的一半加上3呢?加上3是基于数据包守恒来说的，既然已经收到了3个冗余ACK，说明有三个数据分段已经到达了接收端，既然三个分段已经离开了网络，那么就是说可以在发送3个分段了，只要再收到一个冗余ACK，这也说明1个分段已经离开了网络，因此就将拥塞窗口加1个MSS。直到收到新的ACK，说明直到收到第三个冗余ACK时期发送的TCP分段都已经到达对端了，此时进入正常阶段开始加性增加拥塞窗口。

疑难杂症17：超时重传和收到3个冗余ACK后重传

这两种重传的意义是不同的，超时重传一般是因为网络出现了严重拥塞(没有一个分段到达，如果有的话，肯定会有ACK的，若是正常ACK，则重置重传定时器，若是冗余ACK，则可能是个别报文丢失或者被重排序，若连续3个冗余ACK，则很有可能是个别分段丢失)，此时需要更加严厉的缩小拥塞窗口，因此此时进入慢启动阶段。而收到3个冗余ACK后说明确实有中间的分段丢失，然而后面的分段确实到达了接收端，这因为这样才会发送冗余ACK，这一般是路由器故障或者轻度拥塞或者其它不太严重的原因引起的，因此此时拥塞窗口缩小的幅度就不能太大，此时进入快速重传/快速恢复阶段。

疑难杂症18：为何收到3个冗余ACK后才重传

这是一种权衡的结构，收到两个或者一个冗余ACK也可以重传，但是这样的话可能或造成不必要的重传，因为两个数据分段发生乱序的可能性不大，

超过三个分段发生乱序的可能性才大，换句话说，如果仅仅收到一个乱序的分段，那很可能被中间路由器重排了，那么另一个分段很可能马上就到，然而如果连续收到了3个分段都没能弥补那个缺漏，那很可能是它丢失了，需要重传。因此3个冗余ACK是一种权衡，在减少不必要重传和确实能检测出单个分段丢失之间所作的权衡。

注意，冗余ACK是不能捎带的。

疑难杂症19：乘性减和加性增的深层含义

为什么是乘性减而加性增呢？拥塞窗口的增加受惠的只是自己，而拥塞窗口减少受益的大家，可是自己却受到了伤害。哪一点更重要呢？我们知道TCP的拥塞控制中内置了公平性，恰恰就是这种乘性减实现了公平性。拥塞窗口的1个MSS的改变影响一个TCP发送者，为了使得自己拥塞窗口的减少影响更多的TCP发送者-让更多的发送者受益，那么采取了乘性减的策略。

当然，BIC算法提高了加性增的效率，不再一个一个MSS的加，而是一次加比较多的MSS，采取二分查找的方式逐步找到不丢包的点，然后加性增。

疑难杂症20：TCP连接的传输稳定状态是什么

首先，先说一下发送端的发送窗口怎么确定，它取的是拥塞窗口和接收端通告窗口的最小值。然后，我们提出三种发送窗口的稳定状态：

a.IP互联网络上接收端拥有大窗口的经典锯齿状
b.IP互联网络上接收端拥有小窗口的直线状态
c.直连网络端点间的满载状态下的直线状态

其中a是大多数的状态，因为一般而言，TCP连接都是建立在互联网上的，而且是大量的，比如Web浏览，电子邮件，网络游戏，Ftp下载等等。TCP发送端用慢启动或者拥塞避免方式不断增加其拥塞窗口，直到丢包的发生，然后进入慢启动或者拥塞避免阶段(要看是由于超时丢包还是由于冗余ACK丢包)，此时发送窗口将下降到1或者下降一半，这种情况下，一般接收端的接收窗口是比较大的，毕竟IP网络并不是什么很快速的网络，一般的机器处理速度都很快。

但是如果接收端特别破，处理速度很慢，就会导致其通告一个很小的窗口，这样的话，即使拥塞窗口再大，发送端也还是以通告的接收窗口为发送

窗口，这样就不会发生拥塞。最后，如果唯一的TCP连接运行在一个直连的两台主机上，那么它将独享网络带宽，这样该TCP的数据流在最好的情况下将填满网络管道(我们把网络管道定义为带宽和延时的乘积)，其实在这种情况下是不存在拥塞的，就像你一个人独自徘徊在飘雨黄昏的街头一样...

5.2.4.主动的拥塞避免

前面我们描述的拥塞控制方式都是试探性的检测，然后拥塞窗口被动的进行乘性减，这样在接收端窗口很大的情况下(一般都是这样，网络拥堵，分段就不会轻易到达接收端，导致接收端的窗口大量空置)就可能出现锯齿形状的“时间-窗口”图，类似在一个拥堵的北京X环上开车，发送机发动，车开动，停止，等待，发动机发动，车开动...听声音也能听出来。

虽然TCP看不到下面的IP网络，然而它还是可以通过检测RTT的变化以及拥塞窗口的变化推算出IP网络的拥堵情况的。就比方说北京东四环一家快递公司要持续送快递到西四环，当发件人发现货到时间越来越慢的时候，他会意识到“下班高峰期快到了”...

可以通过持续观测RTT的方式来主动调整拥塞窗口的大小而不是一味的加性增。然而还有更猛的算法，那就是计算两个差值的乘积：

(当前拥塞窗口-上一次拥塞窗口)x(当前的RTT-上一次的RTT)

如果结果是正数，则拥塞窗口减少1/8，若结果是负数或者0，则窗口增加一个MSS。注意，这回不再是乘性减了，可以看出，减的幅度比乘性减幅度小，这是因为这种拥塞控制是主动的，而不是之前的那种被动的试探方式。在试探方式中，乘性减以一种惩罚的方式实现了公平性，而在这里的主动方式中，当意识到要拥塞的时候，TCP发送者主动的减少了拥塞窗口，为了对这种自首行为进行鼓励，采用了小幅减少拥塞窗口的方式。需要注意的是，在拥塞窗口减小的过程中，乘积的前一个差值是负数，如果后一个差值也是负数，那么结果就是继续缩减窗口，直到拥塞缓解或者窗口减少到了一定程度，使得后一个差值成了正数或者0，这种情况下，其实后一个差值只能变为0。

疑难杂症21：路由器和TCP的互动

虽然有了5.2.4节介绍的主动的拥塞检测，那么路由器能不能做点什么帮助检测拥塞呢？这种对路由器的扩展是必要的，要知道，每天有无数的TCP要通过路由器，虽然路由器不管TCP协议的任何事(当然排除连接跟踪之类

的，这里所说的是标准的IP路由器)，但是它却能以一种很简单的方式告诉TCP的两端IP网络发生了拥堵，这种方式就是当路由器检测到自己发生轻微拥堵的时候随机的丢包，随机丢包而不是连续丢包对于TCP而言是有重大意义的，随机丢包会使TCP发现丢弃了个别的分段而后续的分段仍然会到达接收端，这样TCP发送端就会接收到3个冗余ACK，然后进入快速重传/快速恢复而不是慢启动。

这就是路由器能帮TCP做的事。

6.其它

疑难杂症22：如何学习TCP

很多人发帖问TCP相关的内容，接下来稀里哗啦的就是让看《TCP/IP详解》和《Unix网络编程》里面的特定章节，我觉得这种回答很不负责任。因为我并不认为这两本书有多大的帮助，写得确实很不错，然而可以看出Richard Stevens是一个实用主义者，他喜欢用实例来解释一切，《详解》通篇都是用tcpdump的输出来讲述的，这种方式只是适合于已经对TCP很理解的人，然而大多数的人是看不明白的。

如果想从设计的角度来说，这两本书都很烂。我觉得应该先看点入门的，比如Wiki之类的，然后看RFC文档,793，896，1122等)，这样你就明白TCP为何这么设计了，而这些你永远都不能在Richard Stevens 的书中得到。最后，如果你想，那么就看一点Richard Stevens 的书，最重要的还是写点代码或者敲点命令，然后抓包自己去分析。

疑难杂症23：Linux，Windows和网络编程

我觉得在Linux上写点TCP的代码是很不错的，如果有BSD那就更好了。不推荐用Winsock学习TCP。虽然微软声称自己的API都是为了让事情更简单，但实际上事情却更复杂了，如果你用Winsock学习，你就要花大量的时候去掌握一些和网络编程无关但是windows平台上却少不了的东西

6.1.总结

TCP协议是一个端到端的协议，虽然话说它是一个带流量控制，拥塞控制的协议，然而正是因为这些所谓的控制才导致了TCP变得复杂。同时这些特性是互相杂糅的，流量控制带来了很多问题，解决这些问题的方案最终又带来了新的问题，这些问题在解决的时候都只考虑了端到端的意义，但实际上

TCP需要尽力而为的IP提供的网络，因此拥塞成了最终的结症，拥塞控制算法的改进也成了一个单独的领域。

在学习TCP的过程中，切忌一锅粥一盘棋的方式，一定要分清楚每一个算法到底是解决什么问题的，每一个问题和其他问题到底有什么关联，这些问题的解决方案之间有什么关联，另外TCP的发展历史也最好了解一下，这些都搞明白了，TCP协议就彻底被你掌控了。接下来你就可以学习Socket API了，然后高效的TCP程序出自你手！

原文链接:
http://blog.csdn.net/dog250/article/details/6612496?utm_source=tuicool

随便扯扯,程序员应该具备哪些素质

作者:星夜落尘

趁着这几天无事，好好总结一下从事软件开发以来的一些想法，这篇blog尝试从我自身的一些经历来谈谈程序员应该具备哪些素质。如有不足之处，还请不吝赐教!

下面，我将列出并展开所有我认为程序员必须具备的素质。

基础知识

你也许是像我一样的自学者，没有数电/模电，编译原理，操作系统原理，网络与数据库等方面的知识，但是对于这些你应该尝试去了解、理解。当初跨专业考研之时学习的操作系统/网络/数据结构/数据库的知识于我现在的工作仍然有益，我有遇到过一些能力很强的人，他们做解决方案很强，但是debug能力说实话不大匹配其水平，原因就在于其不了解很多底层的原理。

对于C/C++ 程序员而言，其底层是操作系统和编译器，所以需要了解操作系统原理，汇编，编译原理等。

对于Java/C# 程序员而言，其底层是虚拟机和框架，也应该去尝试了解虚拟机的构成，GC原理等。

我在网上遇到很多C/C++，Java/C#程序员，很多时候都发现前者更喜欢追根溯底，后者更在乎如何使用框架，无法评判那种态度更好，但是了解得更深入很显然是有好处的。

算法与数据结构

推荐阅读《大话数据结构》，《算法导论》。

算法与数据结构怎么强调都不为过，虽然大部分程序员在工作中并不一定会用到很多高级算法，也并不会去参加ACM，但是理解常用算法应该是一种基本素质。

应该掌握的常用的数据结构：数组，单链表，栈，队列，二叉树。

应该掌握的常用的算法：顺序查找，二分查找；冒泡排序，选择排序，插入排序；深度优先算法，广度优先算法。

进阶数据结构：双链表，循环链表，双端队列，哈希表，跳表，大根/小根堆，哈夫曼树，排序二叉树，平衡二叉树，红黑树，B树/B+树，图，etc。

进阶算法：二叉排序树查找；快速排序，希尔排序，堆排序，归并排序，桶排序，基数排序；KMP字符串匹配算法 ，etc。

自学能力

自学能力对于程序员来说非常重要，因为IT这行更新的太快了，几年不学习很容易就会被时代抛弃，国内尘嚣其上的“30岁转行论”有这方面的原因。我不是科班毕业的，大学学的是水利，和程序员什么关系都没有，一直以来都是自学，我将结合我的经历谈谈自学。

首先需要学习的是编程语言。我接触的第一门语言是Visual Basic，大部分理工科应该都有这样的一门编程课，或是vb，或是c，也有的是c++。这种课程对于大部分不感兴趣的人来说都是一种折磨，不过我对这种计算机按照我的意图来执行非常感兴趣，所以这样的课程很对我的口味。在课余，我会去图书馆借一些vb的书来看看，偶尔也会跟着书上的代码对照着敲代码。后来觉得vb功能有限，且无法理解vb之下的秘密，所以转而自学java，这次是在网上找的马士兵老师的java视频教程，跟着学习了一段时间。因为我的笔记本性能不够，臃肿的JVM运行的非常慢，转而学习C++。现在我在学C++11。

其次要学的是框架，库，API等，这时候你可以尝试做一些有意思的程序出来，通过这些学习基本上可以胜任某些方面的工作。

再次要学的是某个具体的方向，比如web，图形，图像，搜索引擎，机器学习等等专业领域，这些知识的学习应该是在日常工作中不断积累的。

这段时间的自学告诉我，凡事只要去努力去学习，都会有成果，所有看起来高大上的东西理解之后会发现就那么回事。此外，比较广泛的阅读了许多书籍，对我现在的工作仍然有利，许多事情是你首先得了解你才会去应用，很多好东西在书上在网上，你知道了才会在某个未来的时刻用上，如果你不知道那你只能错失良方许久。

学习是持久的，在实际应用中仍然会碰到你不熟悉的特性，会碰到坑，这时候你需要的是信息搜集与筛选的能力!

信息搜集与筛选

在实际编程中，肯定会碰到各种各样的问题，有些是常见的问题，有些是莫名其妙的问题。

我的建议是首先尝试自己解决，次之看官方文档和讨论区是否有解释，然后再去搜索引擎/stackoverflow查找有没有相似问题的解决方案，最后再去社区（CSDN/cnblogs/oschina/stackoverflow等）提问。

强烈建议分享自己的解决方案和思考！

作为一个互联网/ 开源受益者，分享应该是一种基本美德，特别鄙视发布问题自己找到解决方案之后就结贴走人的程序员。

分享自己的解决方案与思考不仅仅是让像你一样的疑惑者受益，同时还是教学相长的一个过程，自己也会从中获益。

如何分享自己的思考，这时候你需要的是总结的能力！

总结能力

总结使人进步！ 在网上看到一个段子，分享一下：

-“你有几年的工作经验，怎么写的代码这么差劲？”

-“5年工作经验”

-“呵呵，是1年经验当5年用了吧。”

决定我们是1年经验当5年用还是真有5年经验，最重要的就是记得总结自己碰到的问题，自己的想法，前辈的教导！

需求分析与文档编写

一个项目的流程大致为：需求分析 –> 估计进度 –> 设计架构 –> 编码实现 –> Debug –> 测试 –> Release 。其中coding,debug,test可能会反复迭代。

可以看出需求分析是决定项目的第一个关键部分，有些公司是由专门人员进行需求分析，但是作为程序员，应该要了解需求，确认需求。

文档包括很多方面，需求文档、设计文档、测试文档、使用文档、注释等，贯穿软件开发的所有流程。良好的文档不仅仅是对项目的负责，同时也会有利于项目的维护。

在项目注释中，强烈建议添加：TODO , FIXME , HACK , XXX 等标签以帮助实现逻辑。

架构能力

在我刚入职的时候，每当接到一个任务时，我都迫不及待的去在IDE中敲代码，这种渴望很强烈，很有成就感。但是一个前辈告诉我，你首先应该做的是架构设计，充分考虑所有可能的情况并记录下来之后再去coding，我记下来了但是在没有教训之前仍然没有很强烈的体悟，后来我便后悔了。在某个项目中，我很快的写出了原型，然后洋洋得意地在这个原型上像打补丁一样扩展各种功能，最后在新加的某个功能上栽了跟头，这个功能完全没办法**凑进去** 。所以，作为程序员，我们需要拟制住自己的编码冲动以及修改代码的冲动，先架构设计，然后再编码。

架构期应该给各个模块/类 之间涉及一套相对合理稳定的接口，实现是易变的，接口不应该频繁变化。

我认为开发任何一个模块，首先要做的是理解需求，然后做架构设计，再然后布置基础设施（包括log，复用的宏，工具代码等），接着进行编码实现。

代码编写

推荐阅读《编写可读代码的艺术》。

这个不用多说，没有代码就没有软件。想追求卓越，应该让我们的代码更优美，性能更好。

代码编写功底包括变量命名，函数拆分与提取，面向对象的特性应用，跨平台意识，多线程的同步，等。

这种能力是在日常coding中积累而来的，多做总结。

Debug能力

推荐阅读《格蠹汇编:软件调试案例集锦》。

没有不出现任何bug的一次性成型的代码，debug是经常会出现的场景。

debug应该尽量的少，同架构设计一样，碰到问题应该首先看代码，能直接找出来问题最好。如果看不出来，就需要专业的debug能力了。

bug的场景包括：逻辑错误，程序crash，内存泄露。

bug的范围包括：单模块，多模块；单线程，多线程；单进程，多进程；单机，联机。

bug的频率包括：100%出现，容易出现，很难出现。

可见debug的范围之广，bug总是如影随形。

很多时候我会抱怨，oh，见鬼了，这太莫名其妙了，在我机器上都不会出现，等等。我现在明白拿到反馈的bug我该怎么做了：首先闭嘴，然后重现问题，接着缩小问题范围，最后借助调试器或者log找出问题原因。

代码阅读能力

说实话这一段我写的很伤感，因为要写“read the fucking source code”实在是太fucking了。

对于接手一个遗留项目，你需要的不仅仅是勇气，还有耐心。对于一个拿到的项目，首先要做的应该是先跑起来，作为用户去使用，了解它的功能；接着阅读设计文档，了解设计意图；再然后如果有版本控制历史的话，可以尝试从早期版本进行代码阅读；再然后是从main函数起大致走一下流程，了解关键route；再然后是对感兴趣部分进行单步深入；最后是通读代码，可以先将功能性代码比如log，hashtable等标记为已读，从头文件看代码间的联系，弄懂各个类/函数的职责。

说实话，对于具有一定强迫症的同学来说，阅读其他人写的代码应该是挺痛苦的，这时候有时间的话不妨对这些代码按你的标准进行重构。

重构能力

推荐阅读《重构,改善既有的代码设计》。

不管是重构别人写的代码，还是重构自己写的代码，好像都不是什么令人愉快的体验。重构，不仅可以使得架构更加合理，而且使得程序更加健壮。

重构，按我的理解分为两种。

一种是自顶向底的重构，这需要对项目具有彻底的理解，才能高屋建瓴的对模块/接口/类/函数进行重新划分，再将以前的代码逻辑填充到新的框架下。

另一种是自底向顶的重构，这种方式是对某些代码进行优化重构，并且保证不影响实现，在重构部分区域之后调整局部结构，最后达到整体重构。

工具的选择与积累

作为程序员，都应该有一套自己使用的得心应手的工具，这样会事半功倍。

这些工具包括：IDE，编辑器，辅助debug的工具，检测系统/程序状态的工具，版本控制工具，文件比较工具，性能分析工具，make/cmake等等。

团队协作

现代软件工程单打独斗基本上不大现实了，像传说中的汇编写wps的求伯君大牛那样独自搞wps的传说已经渐隐渐逝了。

团队协作包括沟通能力，接口协商，版本控制工具的使用等方面。容易出现的是相互推诿责任，对别人的请求不耐烦等，这于团队协作毫无益处。

高效的团队协作应该是模块间接口稳定，基础类库一致，框架代码共享，版本更新信息及时，沟通反应快速有效。

原文链接:

http://www.cnblogs.com/xylc/p/3699388.html

Web基础架构:负载均衡和LVS

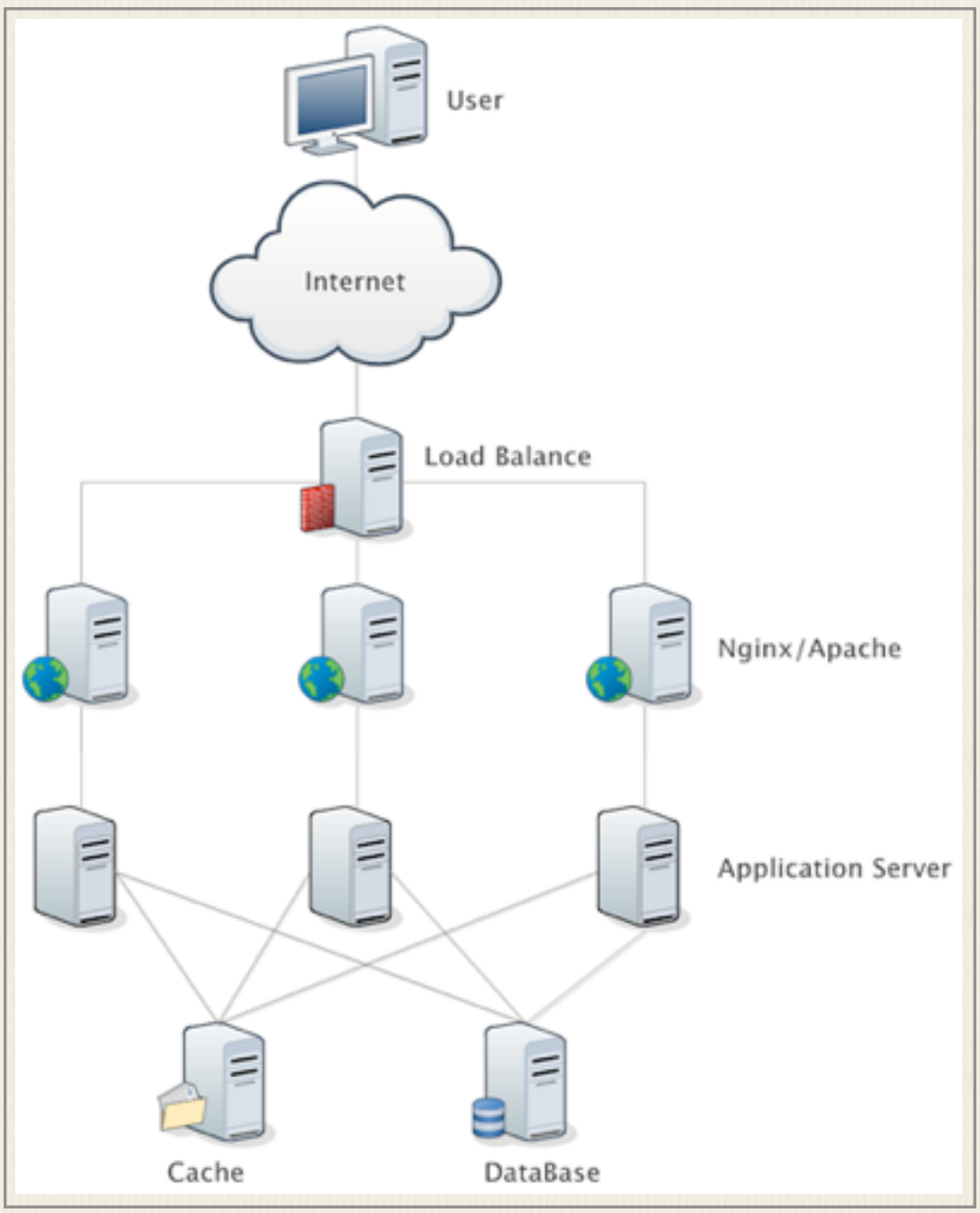
作者:沐剑

在大规模互联网应用中，负载均衡设备是必不可少的一个节点，源于互联网应用的高并发和大流量的冲击压力，我们通常会在服务端部署多个无状态的应用服务器和若干有状态的存储服务器（数据库、缓存等等）。

一、负载均衡的作用

负载均衡设备的任务就是作为应用服务器流量的入口，首先挑选最合适的一台服务器，然后将客户端的请求转发给这台服务器处理，实现客户端到真实服务端的透明转发。最近几年很火的「云计算」以及分布式架构，本质上也是将后端服务器作为计算资源、存储资源，由某台管理服务器封装成一个服务对外提供，客户端不需要关心真正提供服务的是哪台机器，在它看来，就好像它面对的是一台拥有近乎无限能力的服务器，而本质上，真正提供服务的，是后端的集群。

一个典型的互联网应用的拓扑结构是这样的:



二、负载均衡的类型

负载均衡可以采用硬件设备，也可以采用软件负载。

商用硬件负载设备成本通常较高（一台几十万上百万很正常），所以在条件允许的情况下我们会采用软负载，软负载解决的两个核心问题是：选谁、转发，其中最著名的是LVS（Linux Virtual Server）。

三、软负载——LVS

LVS是四层负载均衡，也就是说建立在OSI模型的第四层——传输层之上，传输层上有我们熟悉的TCP/UDP，LVS支持TCP/UDP的负载均衡。

LVS的转发主要通过修改IP地址（NAT模式，分为源地址修改SNAT和目标地址修改DNAT）、修改目标MAC（DR模式）来实现。

那么为什么LVS是在第四层做负载均衡?

首先LVS不像HAProxy等七层软负载面向的是HTTP包，所以七层负载可以做的URL解析等工作，LVS无法完成。其次，某次用户访问是与服务端建立连接后交换数据包实现的，如果在第三层网络层做负载均衡，那么将失去「连接」的语义。软负载面向的对象应该是一个已经建立连接的用户，而不是一个孤零零的IP包。后面会看到，实际上LVS的机器代替真实的服务器与用户通过TCP三次握手建立了连接，所以LVS是需要关心「连接」级别的状态的。

LVS的工作模式主要有4种:

DR

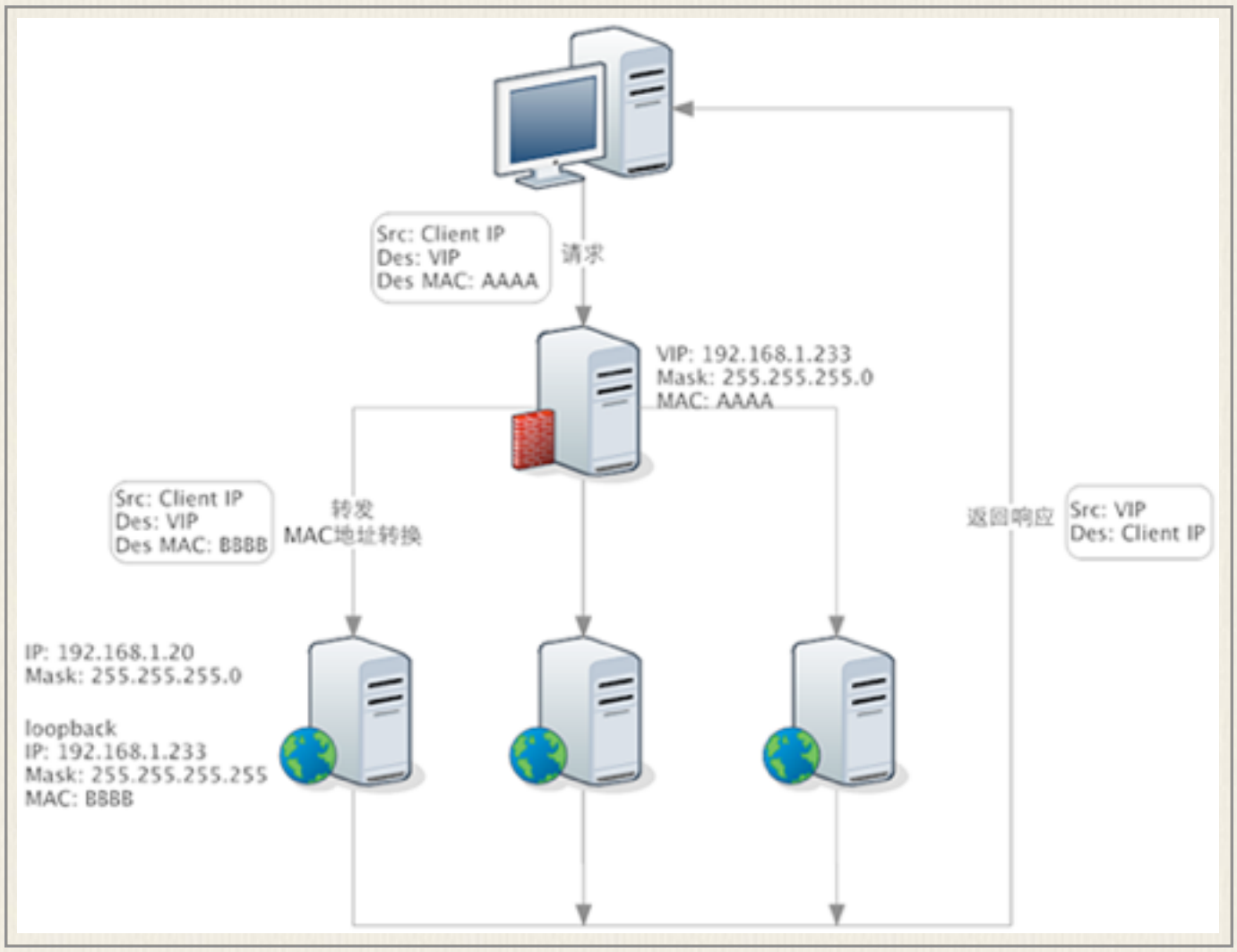
NAT

TUNNEL

Full-NAT

这里挑选常用的DR、NAT、Full-NAT来简单介绍一下。

1、DR



请求由LVS接受，由真实提供服务的服务器（RealServer, RS）直接返回给用户，返回的时候不经过LVS。

DR模式下需要LVS和绑定同一个VIP（RS通过将VIP绑定在loopback实现）。

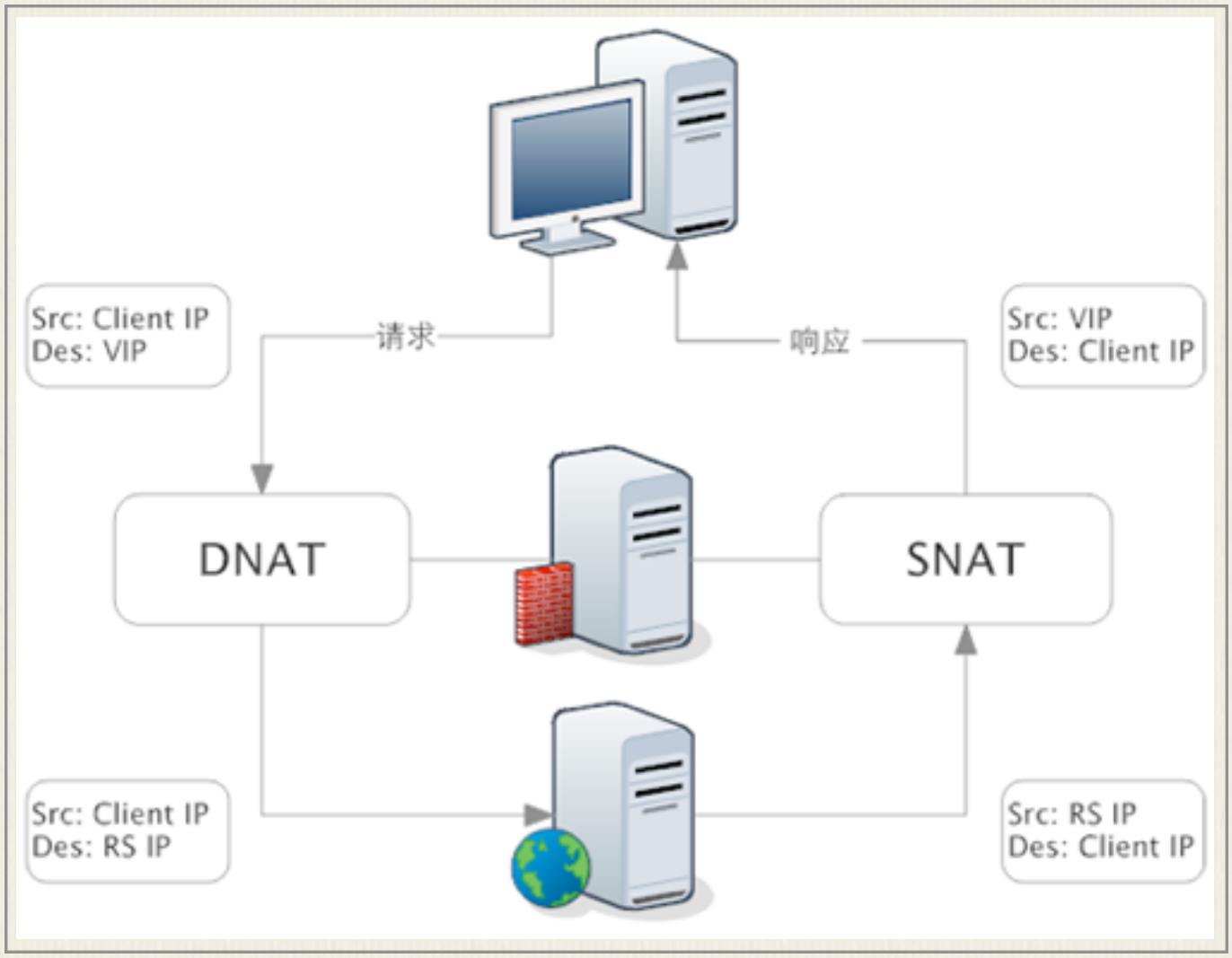
一个请求过来时，LVS只需要将网络帧的MAC地址修改为某一台RS的MAC，该包就会被转发到相应的RS处理，注意此时的源IP和目标IP都没变，LVS只是做了一下移花接木。

RS收到LVS转发来的包，链路层发现MAC是自己的，到上面的网络层，发现IP也是自己的，于是这个包被合法地接受，RS感知不到前面有LVS的存在。

而当RS返回响应时，只要直接向源IP（即用户的IP）返回即可，不再经过LVS。

DR模式是性能最好的一种模式。

2、NAT



NAT（Network Address Translation）是一种外网和内网地址映射的技术。

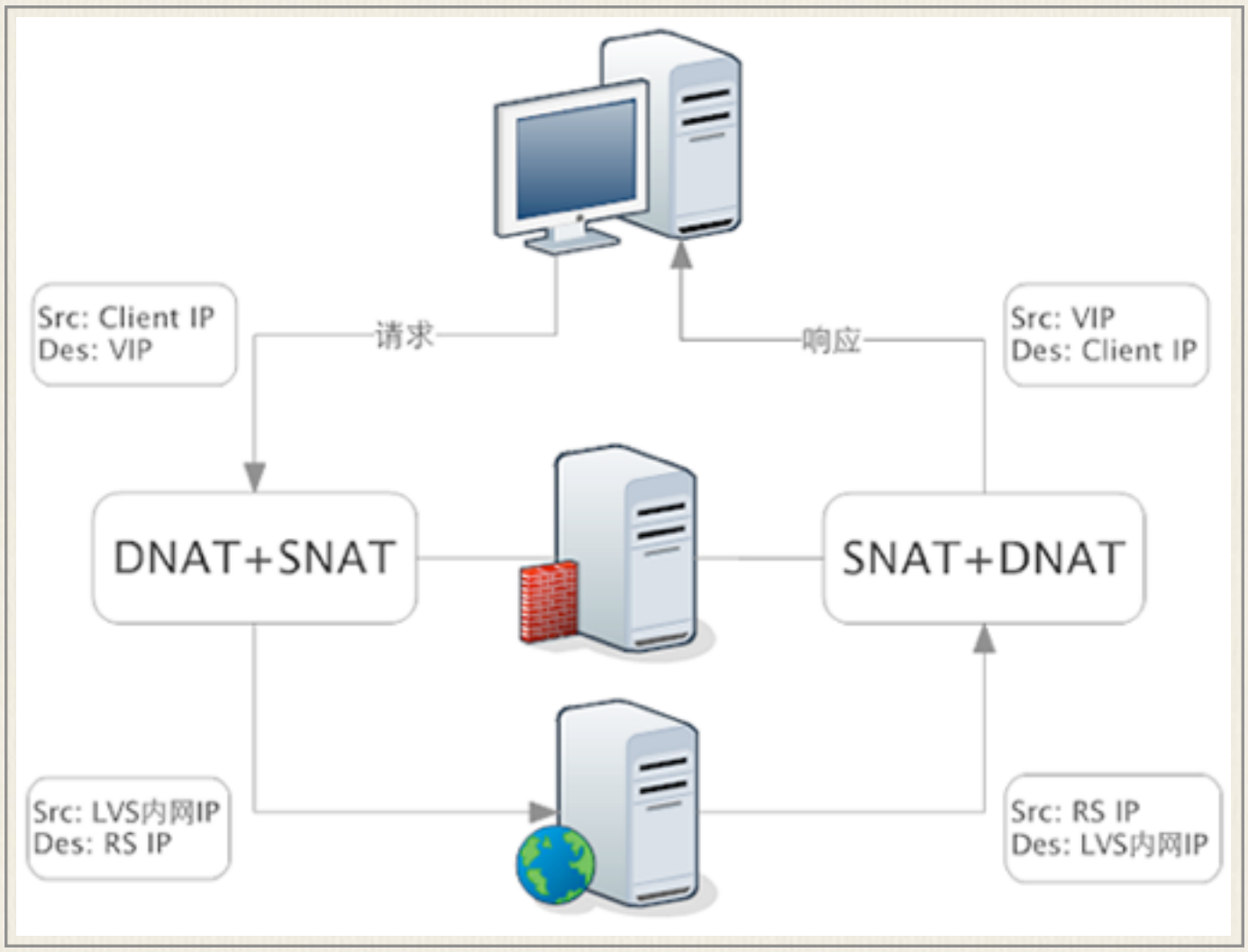
NAT模式下，网络报的进出都要经过LVS的处理。LVS需要作为RS的网关。

当包到达LVS时，LVS做目标地址转换（DNAT），将目标IP改为RS的IP。RS接收到包以后，仿佛是客户端直接发给它的一样。

RS处理完，返回响应时，源IP是RS IP，目标IP是客户端的IP。

这时RS的包通过网关（LVS）中转，LVS会做源地址转换（SNAT），将包的源地址改为VIP，这样，这个包对客户端看起来就仿佛是LVS直接返回给它的。客户端无法感知到后端RS的存在。

3、Full-NAT



无论是DR还是NAT模式，不可避免的都有一个问题：LVS和RS必须在同一个VLAN下，否则LVS无法作为RS的网关。

这引发的两个问题是：

1、同一个VLAN的限制导致运维不方便，跨VLAN的RS无法接入。

2、LVS的水平扩展受到制约。当RS水平扩容时，总有一天其上的单点LVS会成为瓶颈。

Full-NAT由此而生，解决的是LVS和RS跨VLAN的问题，而跨VLAN问题解决后，LVS和RS不再存在VLAN上的从属关系，可以做到多个LVS对应多个RS，解决水平扩容的问题。

Full-NAT相比NAT的主要改进是，在SNAT/DNAT的基础上，加上另一种转换，转换过程如下：

在包从LVS转到RS的过程中，源地址从客户端IP被替换成了LVS的内网IP。

内网IP之间可以通过多个交换机跨VLAN通信。

当RS处理完接受到的包，返回时，会将这个包返回给LVS的内网IP，这一步也不受限于VLAN。

LVS收到包后，在NAT模式修改源地址的基础上，再把RS发来的包中的目标地址从LVS内网IP改为客户端的IP。

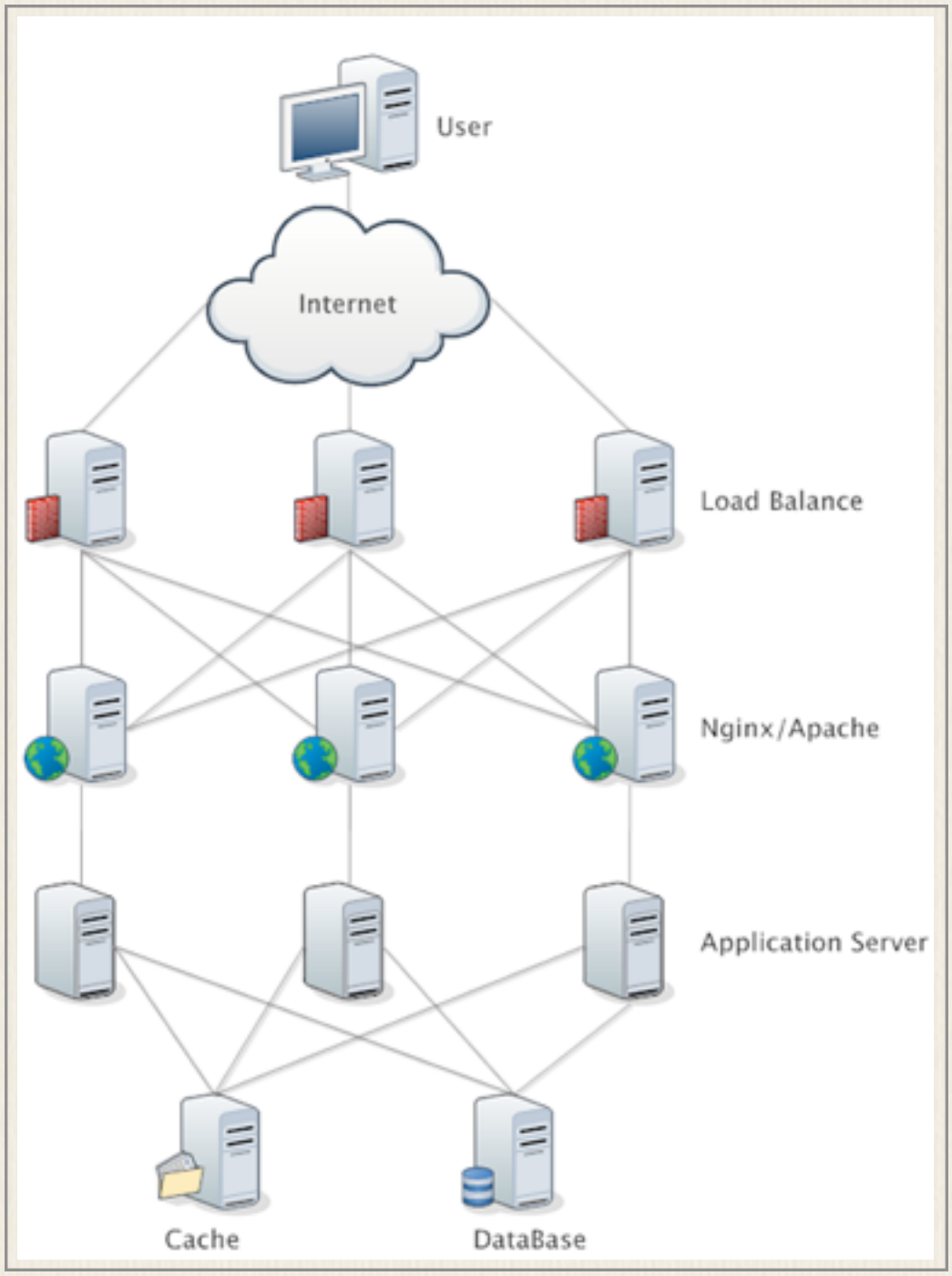
Full-NAT主要的思想是把网关和其下机器的通信，改为了普通的网络通信，从而解决了跨VLAN的问题。采用这种方式，LVS和RS的部署在VLAN上将不再有任何限制，大大提高了运维部署的便利性。

4、Session

客户端与服务端的通信，一次请求可能包含多个TCP包，LVS必须保证同一连接的TCP包，必须被转发到同一台RS，否则就乱套了。为了确保这一点，LVS内部维护着一个Session的Hash表，通过客户端的某些信息可以找到应该转发到哪一台RS上。

5、LVS集群化

采用Full-NAT模式后，可以搭建LVS的集群，拓扑结构如下



6、容灾

容灾分为RS的容灾和LVS的容灾。

RS的容灾可以通过LVS定期健康检测实现，如果某台RS失去心跳，则认为其已经下线，不会在转发到该RS上。

LVS的容灾可以通过主备+心跳的方式实现。主LVS失去心跳后，备LVS可以作为热备立即替换。

容灾主要是靠KeepAlived来做的。

更多参考资料：吴佳明 – LVS在大规模网络环境下的应用 – O’Reilly Velocity China 2012

原文链接:

http://ifeve.com/web-base-architecture-loadbalance-and-lvs/

开源监控系统中 Zabbix 和 Nagios 哪个更好?

作者:张瑞

我比较看好zabbix这款监控软件，理由如下:

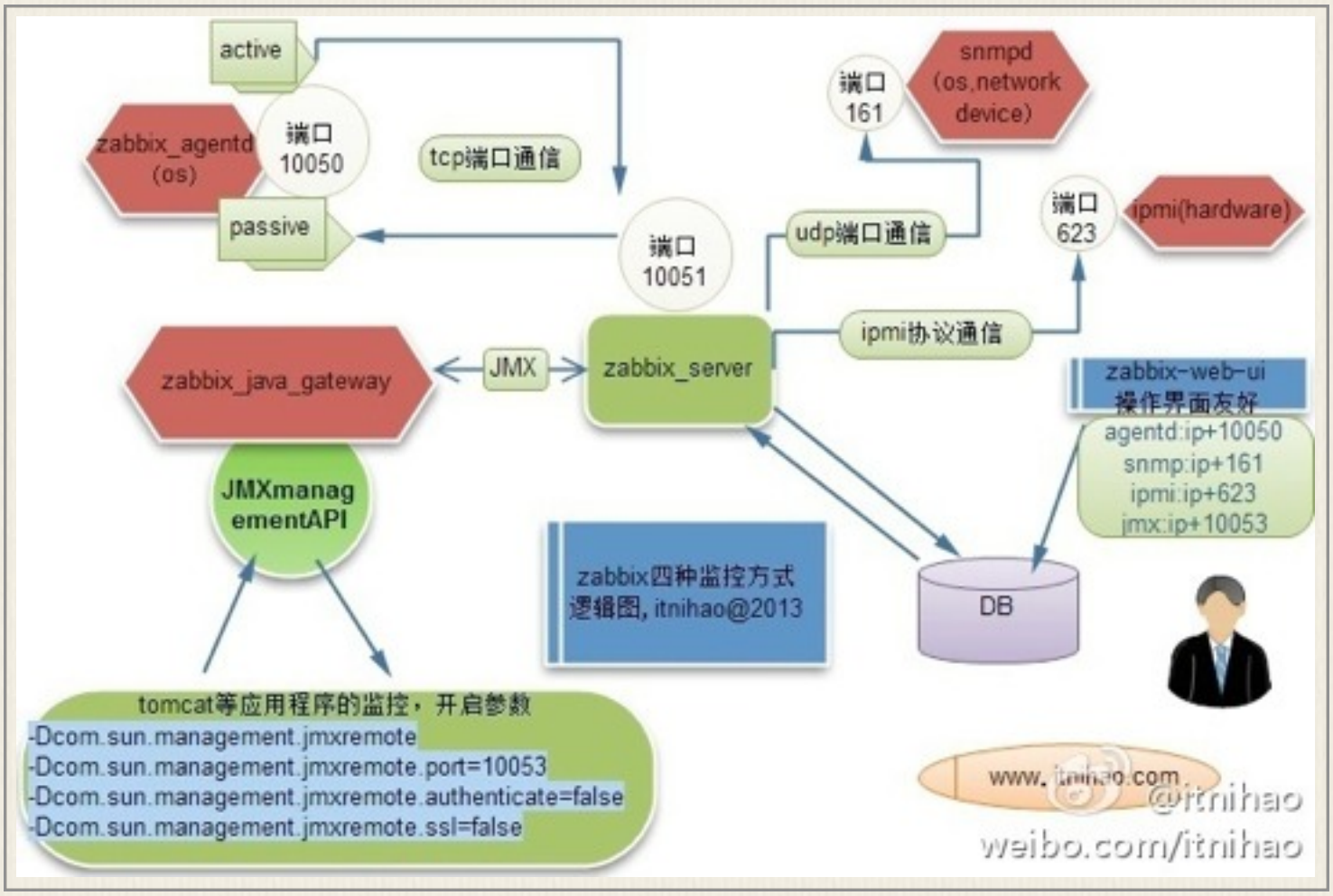
1.分布式监控，天生具有的功能，适合于构建分布式监控系统，具有node，proxy2种分布式模式

2.自动化功能，自动发现，自动注册主机，自动添加模板，自动添加分组，是天生的自动化运维利器的首选，当然于自动化运维工具搭配，puppet+zabbix，或者saltstack+zabbix，那是如鱼得水。

3.自定义监控比较方便，自定义监控项非常简单，支持变量，支持low level discovery，可以参考我写的文档自动化运维之监控篇---
利用zabbix自动发现功能实现批量web url监控

4.触发器，也就是报警条件有多重判断机制，当然，这个需要你去研究一下，这也是zabbix的精华之处，

5.支持多种监控方式，agentd，snmp，ipmi，jmx，逻辑图如下



6.提供api功能，二次开发方便，你可以选用zabbix来进行二次深度开发，结合cmdb资产管理系统，业务管理系统，从而使你的自动化运维系统达到新的高度。

7.当然zabbix还有很多其他功能，这里不一一介绍了。

很多人说zabbix不简单，其实是zabbix的设计理念有些超前，当你都研究到一定程度，你不得不佩服zabbix的团队是非常强悍的，这种工作机制也是相对先进的。

国内的大厂，都有一套自己的监控系统，自己设计，自己开发，其功能也能和zabbix一样，更能适合于自己的需求，但一般企业用，特别是中型互联网公司，还是极力推荐zabbix。

另外附上我的文档Zabbix使用手册V1.4.pdf，这里面有我的经验总结，以及一些使用心得与技巧

最后建议大家多看官方文档

新浪微盘下载地址：最新文档版本为Zabbix使用手册V2.0.pdf

百度网盘下载地址：Zabbix使用手册V2.0.pdf_免费高速下载

同时提供zabbix的安装二次定制的RPM包，该项目地址为：

https://github.com/itnihao/zabbix-rpm/tree/master/zabbix-2.2.2

原文链接:

http://www.zhihu.com/question/19973178/answer/19666150